AN INVESTIGATION OF THE SYMMETRIC PROPERTIES OF LOGICAL FUNCTIONS

ROBERT M. BARR, JR.

UNITED STATES
NAVAL POSTGRADUATE SCHOOL

THESIS

AN INVESTIGATION OF THE SYMMETRIC PROPERTIES OF LOGICAL FUNCTIONS

by

Robert M. Barr, Jr.

Lieutenant Commander, United States Navy

12ND P2238 (1-59)

AN INVESTIGATION OF THE SYMMETRIC PROPERTIES OF LOGICAL FUNCTIONS

* * * *

Robert M. Barr, Jr.

AN INVESTIGATION OF THE SYMMETRIC PROPERTIES OF LOGICAL FUNCTIONS

by

Robert M. Barr, Jr.

Lieutenant Commander,

United States Navy

Submitted in partial fulfillment of
the requirements for the degree of

MASTER OF SCIENCE

IN

ENGINEERING ELECTRONICS

United States Naval Postgraduate School
Monterey, California

1960

AN INVESTIGATION OF THE SYMMETRIC PROPERTIES OF LOGICAL FUNCTIONS

by

Robert M. Barr, Jr.

This work is accepted as fulfilling
the thesis requirements for the degree of
MASTER OF SCIENCE
IN
ENGINEERING ELECTRONICS
from the
United States Naval Postgraduate School

ABSTRACT

The desired response of a logical network can be expressed as a one column matrix of bistable elements, defining the "function" of the network. To date methods of logical network synthesis and simplification have been concerned with manipulations of the network inputs. This investigation treats with the network design as a manipulation of symmetrical properties of the function itself. It will be shown that this treatment not only leads to a logical expression for the network configuration, but can also provide a clue to desirable properties of circuit elements not yet designed. As developed, the network representation will take the form of conventional and-or gating plus a postulated complementing device.

The investigation was performed at Litton Industries, Canoga Park, California, during the period January to March, 1960, while the author was a student at the U. S. Naval Postgraduate School, Monterey, California.

The writer wishes to express his appreciation for the assistance and encouragement given him by Professors Mitchell L. Cotton and John E. Turner, Jr. of the U. S. Naval Postgraduate School and by J. Robert Logan of Litton Industries in this investigation.

TABLE OF CONTENTS

1. Introduction

Modern Weapon systems are demanding the development of larger, more complex date handling and data processing equipments capable of collecting, evaluating and acting on large quantities of information. As a result of these requirements the processing equipments themselves are presenting design problems similar to those originally bringing them into existence, i.e. processing large amounts of data to permit decision making. The logical designer is being forced more and more from a creative position into one of performing a large number of rather menial functions. From a practical viewpoint it is apparent that methods must be developed which can shift the major burden of the design details from humans to automated[1] programs. Of the machines available to assist in such a program the flexibilities, capacities and speed of large scale digital computers make them the natural tools to employ. Some progress has already been achieved in providing time saving techniques to unburden logical designers, but most of the problems require clearer formulation before machine methods can be employed.

Litton Industries Programming Research Group is actively engaged in investigations directed to extend this concept of machines designing machines. More specifically present research studies are concerned with the transformations which take place when the defining mathematical equations are carried into logical expressions. Since

[1] Combination of computers and machines

these expressions provide specifications for final hardware configurations, methods are being sought which would permit the logical designer to force the form of the final networks. However, before processors or computer programs can be devised we must learn more about the transformation process in general, the constraints involved in logical instrumentation and the properties of the system which influence the hardware configurations.

The first part of this thesis presents the results of a review to determine whether or not existing transformation or logical reduction techniques can be applied to the mechanization program. This includes a summary of the basic techniques which have already been used in automatic methods as well as those which are of a more theoretic nature. We will be able to arrive at a set of constraints which should be included in the logical design process and determine whether or not present methods are capable of handling these constraints. It is because we anticipated that present techniques are inadequate and cannot be used for general mechanization that an independent investigation has been undertaken to study a different property of the desired output response functions. A basis is established for defining the symmetric characteristic of these functions and processing techniques are developed which enable us to apply the results as hardware specifications. Finally an unrefined technique for marrying the symmetrical properties with other processes is developed which in special cases provides simplified logical expressions. The final reduction is offered without specifying a reduction criterion.

The present studies have been limited to the parallel operation concept with only brief mention of the application of time-sharing logical elements.

2. The process of designing logical systems

It is intended that this section provide a description in general terms[1] of the requirements placed on a typical logical design group. In addition, it will be shown how the use of computing machines is being broadened to cope with the innumerable details involved. Three distinct steps are required for the presentation of the development. They are:

A. System concept

B. System design

C. System construction

A. System concept

In the beginning, some real need must exist. This might be a requirement to have a large tactical data system, an automatic control system to operate traffic lights in a large metropolitan area, or automation in an industrial application. Whatever the requirement the "system analysis group" will represent the problem exactly by a set of mathematical equations, usually containing numerous continuous or transcendental functions. Since we are really approximating a natural problem in a relatively discreet world, by continuous functions, the initial equations will contain some inherent error. In his attempt to simplify the problems associated with the design, and for economic reasons, the systems analyst will now create another, simpler set of approximations presumably still within the limits of his specifications.

---

[1] For a more detailed treatment from a different point of view, the reader should refer to reference 15 of the Bibliography. M. Phister carries through the details of an approach to this problem.

This reduced set of mathematical equations may still contain some transcendental functions, but is now far more compatible with the intended processing equipment. It is expected that at all times the latest "state of art" is a governing feature.

At this stage the necessity of representing these functions in a form suitable to digital operation may have introduced of itself another genus of errors. This is the type of error which emerges from round-off and truncation at series cut-off points. One must remember that the equations which make up the final set are the equations with which the logical designer must deal. Note that we have accumulated a combination of errors from:

1. The initial statement of the problem by a group of transcendental equations.
2. Fitting the problem to the tolerance of the specifications[1]
3. Computational noise.

Although the individual errors may themselves be small, the combination might well place us beyond the limits of the system tolerance.

In order to determine the suitability of his final system of equations, the systems analyst resorts to a process of Analytic Simulation which is a study of the mathematical characteristics of the proposed system. These characteristics exist in the form of the final defining equations and the constraints imposed by the necessary

---

[1] It would be unreasonable to design a system to provide accuracies within one tenth of one degree if accuracies of one degree were acceptable.

accuracies demanded in the system. The results of Analytic Simulation may be obtained by computer techniques which produce error curves showing nearness of fit, parameter evaluations, or even implicit suggestions as to re-statement of the analytic circumstances. Only when he has such evidence as to the validity and consistency of his expressions can the system analyst be sure of his efforts.

B. System Design

When the analysis group has developed the mathematical equations the logical designer has a foundation on which to build the detailed system design. During the design process he must operate in close contact with programmers and component engineers to maintain system continuity and an awareness of constraints imposed on the design. He must also be capable of utilizing the latest state of art developments.

In recent years more organizations are becoming conscious of the need to have the programmer and logical designer work together. This results from the fact that even though digital processing devices are designed for special purposes and are referred to as "special purpose machines", they are actually general purpose machines operating under a fixed program. Therefore, this close tie between programmer and logical designer must exist to insure an optimum balance between portions of the design which will be programmed and those which will be accomplished by logic alone. This, in itself, represents a vast area open to research and in the eventual scheme should be considered as one of the constraints in the logical design mechanization problem.

Finally, having applied some form of methodological approach to their problem and the existing constraints, the logical designer-programmer team will arrive at a balance between program and a set of simple arithmetic operations which can eventually be represented by logical equations.

Before proceding, a second method of simulation, which we will call Functional Simulation, can be brought to bear at this point to ascertain the realizability of the design. Functional Simulation might be likened to imposing a set of input conditions upon a "black box", capable of performing a set of specified functions, and examining the output. Thus the simulator acts like a transform defined by the operations we wish to perform but arranged such that the internal steps might be obscured.

As with the Analytic Simulation, we resort to the general purpose computer to perform the functions of the black box. Data representing designed program orders is fed into the computer by standard tape or card methods after the computer has been correctly programmed. Several categories of programs exist for setting up, or conditioning[1] computers for functional simulation; but the most frequently used is the interpretive program [1] . When functional simulation has been completed it is possible to decide whether or not the combination of program and logic is suitable. In addition, the logical designer can examine the details of the computer's memory and determine how the simulation was carried through. This may provide the

---

[1] A setting up process by which we make the computer respond to stimuli in the same manner as the transform of the black box to its input variables.

basic building blocks required to develop a logical network for the actual system.

Regardless of the amounts of information supplied by the simulation process the logical designer must still call on his own memory, and by ingenuity and mental correlation arrive at a system of logical equations defining the system. The following sections will reflect the inadequacies of the automatic processes currently available to the logical designer which enables him to determine whether or not the system of equations are expressed in simplest form with regard to imposed constraints. At most, it should be evident that the road to arrive at the end equations can be a long,tedious one, full of indecision, and little assurance at the end whether or not the equations truly represent the system.

A powerful tool, not in general use, but available, is the third form of simulation, Operational Simulation. This was developed to provide a means by which the system represented in the form of a set of logical equations might be studied. The Operational Simulator is a computer program[1] capable of providing an output which displays the state of each logical component (flip-flops, amplifiers, etc.) for each clock pulse over a completed system of logical equations. As such, the logical designer can use this material to determine whether or not the logic has given the desired systems responses, and if not, where breakdown has occurred. In this way the design is continuously updated until the final logical network has been determined.

---

[1] The one being considered here is a program developed by the Programming Research Group of Litton Industries for use with the IBM 650 computer.

C. System Construction

It is interesting to note that mechanization has been carried beyond this stage in some of the more recent problems to provide through the use of programs actual wiring instructions to be used on the assembly line. As it stands, the general method of determining the wiring lists involves many manual and off-line machine processes, and in the end there is no complete assurance that an optimum wiring pattern has resulted. This area represented the largest single bottleneck in the recent past and as a consequence received a relatively large amount of attention. Improved techniques have sufficiently relieved the pressures of the wiring problem to enable a re-aportionment of effort to the immediate problems associated with the development of reduced logical equations.

A final step in the process which bears mention will be the evolution of maintenance and check-out procedures. In general, this is another aspect of the operational simulator output which can be regarded as an absolute reference for hardware comparison. This can be made a part of the process whereby wiring instructions are obtained, but for the present it is sufficient to consider it as another constraint which might eventually be imposed upon the mechanized process.

3. Foundation for logical reduction.

Having considered the problems associated with the overall digital design process, we should now examine the means available to the logical designer by which he can effect some reduction of the equations representing the processing equipment. This will form a basis for later discussions concerning the inadequacies of these methods, and the characteristics we hope will be a functional part of future methods. However, before describing the techniques themselves it is necessary that we establish the mathematics used to define digital networks.

These networks could be instrumented in a number of ways. For example, we might remain in our familiar number system based on the radix ten and require that the logic be capable of operating at ten separate, well defined levels. This obviously leads to a host of engineering problems; and, although some systems have actually been constructed to operate in this manner, the demands for greater and greater reliabilities has resulted in nearly exclusive use of the binary system. In other words, the current state of the art permits us to construct near perfect binary elements such as relays and flip-flops, whereas ten level elements, or configurations designed to provide ten levels, are somewhat inferior. Therefore, since it is possible to represent any decimal number by a combination of the ones and zeros from the binary number system, the trend has been to use binary levels in all logical design.

The basic operations of "and" and "or" symbolized by (+) and (·) respectively can be summarized by the following tables:

+	0	1
0	0	1
1	1	1

·	0	1
0	0	0
1	0	1

In addition to these operations we shall make use of the operation prime (') defined by

$$0' = 1$$

$$1' = 0$$

Having established the elements which will make up our language, we should next expand its use to permit conversation. In the logical design process this can be done via the medium of logical equations or by more complicated pictorial methods.

In 1854, George Boole, an English Mathematician, published his now classic book, "An Investigation into the Laws of Thought", on which are founded the mathematical theories of Logic and Probabilities [2] . During this work, he developed a "logical algebra" which has subsequently led mathematicians to develop several new areas of mathematics. Two of these are "Propositional Calculus" and "The Algebra of Classes". The algebra now used in the design of logical networks is commonly referred to as "Boolean Algebra" since it too has evolved from Booles logical algebra.

Boolean algebra was first brought to bear on problems which concerned themselves with the design of relay switching circuits

by C. E. Shannon in 1938 [3] while he was working as a research assistant at the Massachusetts Institute of Technology. Because of the analogous relationship existing between the actions of relays and those of transistors, the same techniques are now being used in the design of present day logical circuits.

There are several advantages in having a mathematical technique such as the Boolean algebra which can be used to represent the switching circuits of a logical design. It is much easier to represent a circuit symbolically as a set of equations rather than by schematics or logical diagrams. Also, being an algebra, a set of postulates[1] have been developed from which rules and identities have become available for the manipulation of the Boolean expressions. Thus, it is possible by proper manipulations to reduce or simplify expressions initially developed by the logical designer, and as we started out to show, it is possible to formulate the network and reduce the logical equations by formal reduction techniques. However, as the number of inputs and outputs is increased, the complexity of these equations increases even faster, and it soon becomes almost impossible to carry out the indicated operations with existing techniques.

Now that it has been established that we will be dealing with the binary number system and Boolean algebra through its identities it is possible to define the expressions generated by logical design. We note that the expressions describing the

[1] See Appendix A

desired output results from some form of truth table representing all possible configurations of the input variables. These expressions, or functions, as they will be defined, can be used to write directly the logical equations which completely describe the output. Having been written directly they contain all the redundancies of the truth table and of the function, as well as redundancies which result from forbidden states. Looking at the simplest form of truth table, that for two variables "a" and "b", we can now develop the equations representing the functions F and F'.

b	a	F	F'
0	0	0	1
0	1	1	1
1	0	0	0
1	1	0	0

Consider the case where there is an output only if the variables assume the configuration 01, it is apparent that the variable "a" must be in the high or "true" state while "b" is in the low or "false" state. There are no other configurations for which an output is desired. Therefore, we might write directly that F is equal to "a" and "b'", where "a" implies that "a" exists in the true state and "b'" implies that "b " is false. More compactly, the "logical equation" might be written,

$$F = ab'$$

If the function F contained additional ones, more terms would result (e.g. $F' = a'b' + ab'$). Hence we have inductively defined a logical equation to be a mathematical equality which represents the

desired output function in terms of the sum of products of the input variables. This is the "canonical form" [4] of the function expressed as the sum of minterms.

It should be pointed out that it is also possible to express the function as a product of maxterms. In this case F and F' become:

$$F = (a + b) \quad (a + b') \quad (a' + b')$$

$$F' = (a + b') \quad (a' + b')$$

However, all reduction processes which have been widely accepted [4,5,7,8] restrict themselves to manipulation of minterms since they are somewhat easier to handle.

Applying some of the Boolean identities we immediately observe that the minterm expression of F' might be reduced and merely written as b'. This suggests that we look for a general technique for carrying out the reduction of expressions resulting directly from truth tables and to formulate the process.

The problem immediately arises concerning the necessary criterion to apply in order to ascertain what constitutes "reduction" or "simplification". This is where we might consider all of the individual function, component, system and engineering constraints, to determine the manner in which they should contribute to the final logic. From these studies, a set of rules could be formulated which might be applied mechanically in the reduction scheme. Unfortunately, what is postulated here has not been done, but rather the resultant criterion have been very basic. This is most certainly true of the mechanizable schemes. The more complex

constraints are generally, if at all, applied through the genius of the logical designer.

4. Boolean logical reduction techniques

Now, let us consider what can be done to reduce logical equations. Several well known techniques have been advanced which have in common the ability to produce a reduced logic using the diode "and" and "or" circuits as the basis for minimization. The three principal reasons offered for having resorted to these criteria are:

(1) Diode circuits are widely used in digital circuit design.

(2) It is possible to describe a straight-forward and practical procedure for arriving at a second-order[1] expression of a given function which is simpler (or as simple as) any other second-order expression for this function in terms of the number of diodes used.

(3) They might be easily compared cost-wise with similar circuits.

Methods using the above criteria as a basis for determining a reduced logical expression can be categorized into three groups.

A. The trial and error method of simplification

B. W. V. Quine method [5, 6] of simplification

C. Methods based on the Quine Technique of simplification

A. Trial and Error Method

The effectiveness of trial and error methods depends largely on the knowledge, judgment and ingenuity with which the logical designer is able to apply the Boolean postulates and identities. As the name

---

[1] The order of the expressions or level of the logic as it is more commonly called is determined by the number of "and" and "or" gates through which a given signal must pass.

implies, no hard and fast set of rules has been established by which the indicated manipulations might be performed. Also, after simplification there is no assurance that the result is truly a minimal two-level logic. In most cases, it is not obvious that reduction is even possible without first rearranging the expressions into a form which suggests application of certain identities. The following examples illustrate why such a procedure would be almost impossible to systemize.

EXAMPLE ONE. Simplify $F = bc'd + b + ab'd' + b'c'd + ab'c$

Rearranging and combining the first and fourth terms in accordance with $a'b + ab = b$, we have

$$F = c'd + b + ab'd' + ab'c$$

Applying the identity $a + a'b = a + b$ to the last three terms gives a further reduction to

$$F = c'd + b + ad' + ac$$

Next using $ab + ac = a(b + c)$ and inverting

$$F = c'd + b + a(c'd)'$$

Again applying $a + a'b = a + b$ and rearranging, we finally find that

$$F = a + b + c'd$$

EXAMPLE TWO. Simplify $F = c'd + b'c + bc' + cd'$

Besides the methods already discussed some very recent work has been reported by Shreeran Akhyankar [12,13] which represents a significant beginning into the search for a truly "Absolute minimal" Boolean function. One must not be misled however, because the "Absolute minimal" expression, LZ(f), which he seeks is again based on very flimsy starting criterion. He defines the length of the function, L(f), as the number of "and" and "or" operations represented by the final expression. This is little different from using the number of diodes, except that the final equations are not restricted by the levels which the logic can assume.

In his first paper [12], Akhyanhar, using the Algebraic Topology terminology and notation of Urbano and Mueller [4] as well as point set theory, formulates from a set of theorems a means of obtaining the minimal expression of the type Zsps(f) (sum-product-sum). He later extends the work [13] to obtain the absolute minimal (as he defines it) expression LZ(f).

Because of the complexities of the general problem, Akhyankar has been necessarily limited to treating the more trivial cases of cell complexes which possess nice geometric properties; e.g., purity of dimension, i.e., all the basic cells being of the same dimension, connectedness, etc. Thus, the theory of special configurations can be applied. Specifically he deals with the one and two isolated point cases to preclude overlap. This assures us when we define the complexes by point sets E_1 and E_2, that the set defined by $E_1 \cap E_2$ is identically zero, i.e., sets are disjoint.

Although the work does not represent a method per se by which one gets minimal logic it is the first mathematical step toward such a

method. However, regardless of the successes in the directions being taken by mathematicians such as Akhyankar, we must not forger the basic need for a method which considers the problem from a basis more profound than the present, simple starting criterion.

The methods discussed in this section have in common two basic characteristics. First, each method based reduction on the same simplification criterion, and secondly, all require that the initial statement representing the desired output function be either a canonical set of minterms of the type

$$f(x_o, x_1 \ldots . x_{n-1}) = \sum_{i=0}^{2^{n-1}} f_i \, (x_o^{\;j_o}, x_1^{\;j_1} \ldots . x_{n-1}^{\;j_{n-1}})$$

or a canonical set of maxterms,

$$g(x_o, x_1 \ldots . x_{n-1}) = \prod_{k=0}^{2^{n-1}} \varepsilon_k(x_o^{\;j_o} + x_1^{\;j_1} + \ldots . x_{n-1}^{\;j_{n-1}})$$

Where the x_o, x_1 .... x_{n-1} are the input variables and the j_o, j_1 .... j_{n-1} can assume either the primed (') or unprimed state. Henceforth, methods which exhibit these characteristics will be referred to as "Class I" methods. Class II methods will be defined later.

5. Extensions of logical reduction

We have now outlined some of the more salient features of current techniques in the area of logical design. Next, some useful data might be obtained by sampling the industrial application of these techniques.

The inescapable fact is that almost the entire region of logic design per se is still subject to "human ingenuity". The reduction methods, advanced as they might be, are used, to be sure, but only as supporting devices or as minor shortcuts in relatively small switching nets. Only where the largest computing machines are available can we say that true design adis are employed to any large extent and "large extent" here means hardly more than an increase in the range of input variables.

Inquiry into the operational level reveals that the techniques are indeed open to far more improvement before they may be used in a comprehensive sense. The idea of extending logic reduction should not stress "reduction" nearly so much as the compatibility of reduction with the environmental constraints imposed on the designer.

A. Inadequacies of present reduction techniques

In reviewing the methods outlined in section four we note that little progress has been realized in the quest for a system that will produce a "minimum" logical representation for a system of equations. The primary failing of present techniques rests with their inability to cope with the wide range of constraints and the sheer massiveness of the many simplification problems. The tendency has been to treat reduction as a mathematical problem requiring formulation for the sake of reduction alone. This is evidenced by the number of mathematicians now offering theorems and parts of solutions. What appears to be

missing is an appreciation of the part which hardware and other constraints must play within the overall framework of any final solution. These constraints will be enumerated later, but first lets look at the limitations of our "pure reduction" techniques in greater detail.

In order to facilitate design of large networks, a number of computer programs have been assembled which apply our Class I techniques [6, 7, 14, 18] . Some of these programs begin with the actual truth table data and develop from this the canonical sets which must be reduced, while still others require as their input parameters the actual minterms of the desired output function. In either case the programs utilize the expressions describing the input output relationships for the given problem, and automatically minimize into two-level logic. In this way the logical designer is provided with a set of equations, which according to the defining criterion, represent an efficient design for the desired network.

These methods are very useful, but unfortunately the systems described are extremely limited as to the number of input variables with which they are capable of dealing. In addition, the programs themselves have no provisions for applying the numerous function and system constraints. These must be added manually by the logical designer and as a result the solution obtained is far from being optimally unique. Once reduction has proceeded so far by machine methods we have lost many of the redundancies which would permit even greater reduction as the constraints are applied.

Note, too, that these methods all require that we carry the entire truth table in the computer memory. There is still no provision for

effecting reduction on segments of the truth table and then combining the results. With these conditions prevailing it appears that we will be continually faced with the need for computers with larger and larger memories. As an example of the magnitude everyday problems tend to assume consider the 10 by 14 sine function generator designed and constructed by Lincoln Laboratories, MIT [14] . Design of this sine function network begins with an expression containing over 56000 minterms. Most computer storages are incapable of carrying sufficient data to do problems of this magnitude in any straight forward manner. Perhaps a possible solution will be to deal with highly redundant segmented functions which can be recombined after preliminary processing. Carrying the redundancies would provide for the eventual recombination into forms which are minimal in the sense of present techniques.

A further difficulty at present results from our inability to specify the form of the final logical equations. Available hardware and technology are rapidly increasing and it is now possible for logical designers to make greater demands upon the component engineers. Circuits such as the "nor" and "exclusive or" are becoming more common. However, there are still no mechanizable methods, by which we can automatically design them into the system.

B. Logical design constraints.

What do we mean when we say "the constraints imposed on the logical designer" or "the system constraints are"? In general we are considering the limiting factors of nature or of engineering technology, but as we shall see there is the additional connotation of space-time relationships which goes along with the term "constraints"

in logical design and reduction processes. A complete discussion of the full impact and implications of these quantities, and their relative importances would be lengthy and inconclusive. Consequently what follows will be a representative list including those constraining factors which are of greatest immediate concern to logical designers. It is important to remember that there exist no straight forward techniques by which these constraints can be applied in the reduction process.

(1) Overall system requirements beyond the mere representation of a set of mathematical equations but rather extending to include such things as the time lags permitted between the time that inputs are applied and outputs must be available, weight and size limitations, power availability, etc.

(2) Component redundancies or coded parity checking systems considered necessary to insure adequate overall reliability.

(3) Hardware or component capabilities dictated by the current "state of the art", e.g. rise time of flip-flops determines the number of logic levels permitted.

(4) The extent to which we can apply the concepts of time-sharing logical elements. In this area logical designers applying ingenious means have been able to duplicate the functions performed by large networks using only a small number of logical elements coupled with the necessary delay devices. This of course spreads the problem in time.

(5) In complex systems, there should exist methods for exactly locating component failures which introduce malfunction. To accomplish this, additional logic can be designed into the overall

networks thereby permitting the use of "maintenance routines" [15], or in cases where the basic equipment is not a programmable computer the information can be presented in some other form.

(6) To what degree should the processing equipment be a general purpose or special purpose device, i.e. what balance must be maintained between program and logic.

(7) In addition to those constraints which are relatively easy to define, there exists an entire class of "man decisions" which are more far reaching and as such cannot be included as part of the immediate goals. As an example, we have the very basic initial decision which must be made by the system designer concerning whether the program is better suited to analog or digital solution. If it is decided to use a combination of the two techniques, what should the balance be, etc.

C. A Class II system.

So far our study has covered the broad problems of logical design which have generated the need for logical reduction schemes, the characteristics and inadequacies of available Class I methods and finally an enumeration of the constraints which should be made a part of any advanced technique. This study suggests that a new approach to the composite problems of logical reduction is needed. A method which would give the logical designer more control over the form of final networks.

In most cases where we have assembled this amount of data

---

[1] See Appendix B

about a basic problem a number of alternative solutions normally suggest themselves. The present situation is much more difficult. We must of necessity develop the components of an overall theory in the process of gradually evolving the general solution. This can only be accomplished by initiating a series of stabs, based on a priori knowledge of the problems characteristics, into area which show most promise; areas where new philosophies uncovered might suggest the next step. Since the problem is of this type it is even more important that the broader concepts which have been presented in the last few sections be thoroughly appreciated before beginning or continuing the investigation. The probes should be with purpose or goal in mind if not in sight. One such effort which proved successful was that made recently by J. R. Logan [18] at Litton Industries, Canoga Park, California.

Logan's method of congruencies is a partitioning process which treats the output function directly rather than the Class I system approach of first converting to an expression of minterms or maxterms. Methods which exhibit this characteristic of treating the output function directly will be referred to as "Class II" systems. It will become apparent as "symmetries" are discussed in the next section that although Class II methods do not deal directly with the Boolean functions or identities, the operative characteristics of Boolean algebra are still implied.

Partitioning by "congruencies" is accomplished by superimposing portions of the output function on top of like portions of the same function in accordance with the binary divisions of a standard truth table of "n" variables. One part of the function can be

considered as having been displaced in time (vertically) to cover the corresponding portion of the function indicated by those bits of the input variable which are of opposite polarity[1]. As an example consider the three variable counter of Fig. 1(a).

c	b	a	F	
0	0	0	0	X
0	0	1	0	
0	1	0	1	
0	1	1	1	
1	0	0	1	Y
1	0	1	0	
1	1	0	1	
1	1	1	0	

(a)

X	Y	XY
0	1	0
0	0	0
1	1	1
1	0	0

(b)

Figure 1

The arbitrary function F is said to be congruent in "c" to the degree:

0
0
1
0
0
0
1
0

We arrive at this mechanically by first locating the coincident bits in the upper and lower halves of the function, Fig. 1(b), and then literally repeating this XY column. This duplicates the manner in which the "c" variable changes polarity. In passing we might also note that the function has congruencies in "b" and "a" to the degree:

---

[1] The difference between one and zero, or true and false, is analogous signals of different polarities.

a	b	a + b
0	0	0
0	0	0
0	1	1
0	1	1
1	0	1
0	0	0
1	0	1
0	0	0

Observe that the selected function can be completely specified by the "b" and "a" congruencies, therefore the "c" congruencies in this case would be discarded as redundant. The method continues by what Logan calls a readjusting process until a form is developed from which the minimal logic is written directly.

Note that partitioning by congruencies can be likened to the fourier analysis of arbitrary waveforms. The basic function, which in a reduction process is the desired output configuration of the ones and zeros, can be though of as being filtered into its frequency components i.e., the fundamental frequency and a limited number of harmonics. The fundamental frequency of the fourier series would correspond to congruence in the most significant variable, the second harmonic to the next most significant variable, etc. until finally the highest harmonic present would correspond to the least significant variable. Hence the number of congruencies or number of frequencies possible is equal to the number of input variables. This limitation corresponds to the truncation of a fourier series which we realize may introduce error or "noise" equal to the value assumed by the remainder term. A similar phenomena occurs in the partitioning processes. Only in those cases where the initial waveform can be represented exactly by a finite number of harmonics along with the fundamental, which may or may not be present, will

there be an absence of noise. The same is true for congruencies. A separate signal must be picked up to complete the function after the congruencies have been removed from the function. As in the Class I systems, this turns out to be an "essential term" which will necessarily occur in the reduced logical equation.

The paper continues after obtaining the minimal expressions to a treatment of redundancies, sorts based on the frequency of occurrence of the input variables in the function, sorts covering systems of functions, etc., in an attempt to influence reduction along desired directions or in accordance with simple constraints.

Although the successes of Logan's work are in themselves significant, the idea of congruencies suggests a whole series of further investigations. One of these, symmetries, is treated in the following sections. It will be shown that this area was of prime interest because it suggests the possibilities of pressing into use a new form of symmetric hardware.

6. Partitioning by the method of symmetries

In the previous section a Class II technique, Reduction by Partitioning, was described briefly. The term partitioning referred to the process whereby characteristic sub-functions (congruencies) could be separated from the original function and treated independently. Since a "set of congruencies" merely represents one special property of the function we have in effect screened the function in such a manner that it is possible to examine this property in greater detail. This suggests that a reapplication of partitioning under some new set of rules would produce still another set of sub-functions exhibiting some further property of the initial function. This is in fact the case, and one of the group of such possibilities which immediately suggests itself is partitioning into "symmetries", i.e. removing the symmetric properties of the function.

It would be difficult to predict without extensive data whether or not the application of symmetries might lead to further economies in logical design. However, since we will be dealing with symmetric properties, it is reasonable to suspect that our logical networks could be represented almost entirely by some form of symmetric hardware. This in no way implies that the representation as such would be a simplified logic in the sense of present Class I and Class II reduction techniques, but rather offers the possibility of eventual combinations of the two forms of hareware into some simpler design. Although logical designers have already discovered unique uses for symmetric circuits such as the "exclusive or",[1] there is as yet no

---

[1] See appendix B for a circuit developed and used by Litton Industires, Beverly Hills, California

mechanical method for designing these components into networks.

The possibilities of developing such techniques and specifying hardware become more obvious if we examine the possible output configurations represented by the simple two-bit counter of Fig. 2. This table arrangement of the input variables "a" and "b" along

b a	f_0	f_1	f_2	f_3	f_4	f_5	f_6	f_7	f_8	f_9	f_{10}	f_{11}	f_{12}	f_{13}	f_{14}	f_{15}
0 0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
0 1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1 0	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1 1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

Figure 2

with any grouping of the desired output functions, f_0 through f_{15}, is commonly referred to as a "truth table" since it gives pictorially the combinations of the input variables which lead to a "true" output. The input variables "a" and "b" might be generated within the system by flipflops, or alternatively they could be external inputs coming from outside the network or system under consideration. In either case, all possible combinations must be considered. Later we will examine the effects of "forbidden states", combinations of the input variables which cannot occur, upon the symmetries of the output function.

In Fig. 2 the functions f_0, f_6, f_9 and f_{15} all exhibit symmetric properties; folding these functions about their midpoints introduces perfect coincidence of the ones and zeros. Considering the functions more closely we observe that f_0 and f_{15}, although symmetric represent the trivial cases of "never an output" and "always an output"

respectively, while f_6 and f_9 are uniquely represented by

$$f_6 = ab' + a'b = a \oplus b$$

$$f_9 = a'b' + ab = a/b$$

the familiar "exclusive or" circuit and what we will call the "equal or" circuit. The latter name is derived from the fact that an output is produced only when "a" is identically equal to "b" in either the true or false states. The symbol "a/b" should be read "a slash b". When symmetric properties are defined, we will extend the idea of folding to allow folds about additional points in the function. For example, if we permit folds between the first and second bits and the third and fourth bits, functions such as f_3 and f_{12} will be defined as symmetric.

Reflecting on the results of partitioning functions into congruences, we recall that the symmetric quantities f_6 and f_9 were not permitted but instead were covered by combinations of congruent functions or "singles", functions with neither congruent nor symmetric properties. In that system the functions f_3, f_5, f_{10}, and f_{12} were of primary importance because as defined f_3 and f_{12} are congruent in "a" while f_5 and f_{10} are congruence in "b", e.g. considering f_5 and f_{10}, there is perfect coincidence of the ones and zeros of the lower and upper halves of the function when the lower half is displaced upward two bit positions. By restricting the technique to these congruent functions, f_3, f_5, f_{10}, and f_{12}, logic expressions are obtained which are minimal in accordance with the two-level, diode defining criterion. Symmetric functions could not be tolerated under these conditions. Therefore, in order to take advantage of the symmetric properties of the desired output

function a new form of hardware will emerge as the basic building block and as it turns out the quantities f_5 and f_{10}, not being symmetrical, will disappear in favor of f_6 and f_9.

Before defining what we mean by symmetries or developing the rules by which we might mechanically partition functions into symmetries, we should establish some form of shorthand which will enable us to handle the sheer bulk of data presented by truth tables. A notation which has been found most satisfactory is the hexadecimal system of numbers. This is a single variable method of counting from zero to fifteen inclusive. A form of the binary-to-hexadecimal conversion which will be used here is given by Figs. 3(a) and 3(b).

d c b a	Hex character	F
0 0 0 0	0	1
0 0 0 1	1	0
0 0 1 0	2	1
0 0 1 1	3	0
0 1 0 0	4	0
0 1 0 1	5	0
0 1 1 0	6	1
0 1 1 1	7	0
1 0 0 0	8	0
1 0 0 1	9	1
1 0 1 0	A	0
1 0 1 1	B	0
1 1 0 0	C	0
1 1 0 1	D	1
1 1 1 0	E	1
1 1 1 1	F	1
(a)	(b)	(c)

Figure 3

Using this notation, a 4-bit counter (four variables a,b,c and d) and the arbitrary output function "F" given in Fig. 3(c) can be collapsed and written as:

d	c	b	a	F
0	0	3	5	A
0	F	3	5	2
F	0	3	5	4
F	F	3	5	7

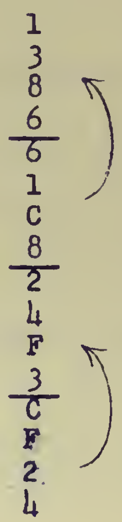
This represents a four-to one visual compression of the data without any loss of significance. If we were to complement the table, the variables would be represented as primes:

d'	c'	b'	a'	F'
F	F	C	A	5
F	0	C	A	D
0	F	C	A	B
0	0	C	A	8

In congruencies, the periodicity of ones and zeros for each variable of an N-bit counter was used as the basis of definition. A function could be congruent to a degree (or in unusual cases entirely congruent) in any or all of the input variables; the variable "a" representing the least significant (2^0) bit position, the variable "b" the next significant (2^1) bit position, etc. However, in treating symmetries, the recursion properties of the separate variables are used somewhat differently. Instead of sliding the groups of bits vertically to cover other groups of bits of equal length and looking for coincidence of the "ones" or true bits between these groups, the groups are now "folded" about a point of symmetry. This point of symmetry is uniquely determined by the periodicities of the input variables. Since the bits of the "a" variable are singly repeated for the length of the column, the folding is performed one bit at a time, i.e. the second bit of the column is folded on to the

first bit, the fourth on to the third, etc. Thus a column or function where the bits are singly repeated for the length of the column is said to be "a function symmetric in a". For example, if we consider the four-bit counter used in Fig. 3(a) the "b", "c" and "d" columns are "symmetric in a". Note especially that "a" itself is not "symmetric in "a".

Next the periodic structure of the "b" column determines the point of symmetry for "b" symmetries"; the bits being folded in pairs about the points where the "b" column of the counter changes from zeros to ones. Note again that the "b" column is not symmetric in "b" itself, but is "symmetric in a". The same criterion for symmetry applies to the "c" and all higher order variable columns as well. The periodic structure, or rate of recurrence of groups of ones and zero bits, determines the point of symmetry about which groups of length 2^{N-1} are folded; N taking on the values 1, 2.....n, where n is the order of the highest order variable present. For example, the following 6-variable function (N=6) has perfect symmetry in the "e" (next most significant digit) variable by reason of the indicated "folds".



In passing, it is important that we observe a special characteristic of the symmetric properties of N-variable counters. In general we say that the variable "a" is "symmetric in nothing", "b" is "symmetric in "a", "c" is "symmetric in a and b", "d" is "symmetric in a, b and c", etc. Therefore each column is symmetric in all lower order variables but not symmetric in the same or higher order variables. This differs basically from the behavior of congruences where each variable of the N-bit counter is congruent in all other variables except the variable of the column being considered.

Assuming that we wish to demonstrate as many properties of functional symmetry as possible, it follows that we might consider examining the function in descending variable order. We might then expect fewer symmetries to show up as we approach the least significant variable. Because we are treating with the function itself rather than truth tables, our manipulations in partitioning are indicative of the actual operation of the end hardware. This unidirectional scan seems to suggest tentatively that symmetrical analysis will lend itself to processing elements which may be distributed serially in time.

Having defined what is meant by the symmetric properties of a given function, we are now prepared to consider the process whereby the functions generated through logical design may be subdivided and examined in greater detail. There is no need to change the form of the original function. Instead it is possible to work directly from the configuration of ones and zeros (stated in hexadecimal notation) determined by the truth table. Since this configuration represents the conditions or values of the input variables which are necessary

to produce the desired outputs, the partitioning process eliminates the need for referring directly to the input variables when processing the data.

As mentioned previously, the partitioning process is being used to determine the symmetrical properties of the function. In the case of congruences if a function happened to be completely congruent in one of the variables, that variable disappeared from the logical expression finally used to represent the function. It turned out that the complete logical definition of a function was no more than a statement of the degree to which the function was not congruent in all of the component input variables. These characteristics were inherent because of the basic definition and will not necessarily apply to symmetries. In fact a re-examination of the F_6 and F_9 functions of Fig. 2, which are completely symmetric in "b", shows that both input variables are necessary to specify either of the functions. Therefore, at this point we have no reason to believe that complete symmetry would imply the disappearance of any of the input variables from the final expression, but intuitively we suspect that the "degree of symmetry" will be useful in deciding the eventual form and simplicity of the final equations.

Before considering these effects further, we must perform the actual partitioning and establish methods for manipulating the symmetric sub-functions. In order that we might have a means of comparing the present methods with those of J. R. LOGAN'S congruences [18], the same function which was partitioned into congruences will now be used to illustrate partitioning into symmetries. Consider the function of Fig. 4 which is 64 or 2^6 bits in length. This

is one of a system of six functions which result from the three by three bit multiplier represented in appendix C, and as it happens is not completely symmetric in any of the six input variables. For this reason it represents a fairly general case.

0
0
0
0
0
F
1
C
3
3
3
6
2
D
2
A

Figure 4

Our first task is to partition this function into the existing symmetries. We begin by looking for symmetry in "f". This is done by folding the functions about the axis of periodicity in "f", which amounts to folding about the midpoint. Once the function is folded the coincidence bits are removed and recorded as in Fig. 5.

upper half	lower half folded upward	coincidence bits
0	5	0
0	4	0
0	B	0
0	4	0
0	6	0
F	C	C
1	C	0
C	C	C

Figure 5

Note the manner in which the lower half of the function must be folded. Since the fold is on a bit by bit basis, the hexadecimal characters from the lower part of the function are not merely re-recorded in inverted fashion, but must be converted to new characters represented by the inverted bits. For example the ninth character "three represents the configuration 0011. When this is folded it becomes 1100, or reading from the top the hexadecimal character "C". Additional care is also necessary when expressing the final symmetry from the coincidence bits. Thus when the coincidence bits of Fig. 5 are unfolded the hexadecimal "C" becomes a "3". Now we can say that the function is symmetric in "f" to the degree:

0
0
0
0
0
C
0
C
3
0
3
0
0
0
0
0

To determine the degree to which the function is symmetric in "e" we refer to the periodicity of "e" in the standard six-variable counter. There is a double change, or two-cycle effect, in this variable, consequently the function must be folded twice to determine symmetries.Fig. 6 illustrates how the function has been divided into quarters and folded, i.e. the second group of four hexadecimal characters is folded upward on to the first four characters and the

last group of four is folded upward on the third group. The third column in each case represents the coincidence bits which indicate the degree of symmetry.

Upper half			Lower half		
Upper	Lower	Coincidence	Upper	Lower	Coincidence
0	3	0	3	5	1
0	8	0	3	4	0
0	F	0	3	B	3
0	0	0	6	4	4

Figure 6

Unfolding the two halves, we can now say that the function is symmetric in "e" to the degree:

0
0
0
0
0
0
0
0
1
0
3
4
2
C
0
8

Partitioning continues in like manner for the remaining input variables. For symmetry in "d", the periodicity is four, consequently the function has four folds. Note that the folds always occur about the points where a standard counter would shift from a zero to one bit. The reader must be cautioned to exercise care when seeking "b" and "a"

symmetries since it is no longer possible to represent the folded parts by hexadecimal characters. In the case of "b" symmetry the folds are made with two bits, and for "a" only one bit. For example, D, which in the hexadecimal notation is 1101, is symmetry in "b" to a degree of "9" and in "a" to the extent of "C". See Fig. 7.

D	folded	coincidence	unfolded
1	1 1	1	1
1	1 0	0	0
0			0
1			1
1	1 1	1	1
1			1
0	0 1	0	0
1			0

Figure 7

Now arranging the symmetric properties of the function into a table or matrix we can determine the extent to which the degrees of symmetry in each of the input variable together provide cover for the initial function. This is shown in Fig. 8. Scanning the symmetries horizontally and applying the "or" proposition gives by our definitions a completely symmetric function; one which does not necessarily have complete symmetry in any single variable, but is symmetric in the combination. In cases where the initial function has these properties the horizontal scan gives back the function identically. However, the present example represents a more general case as far as symmetries are concerned. Here part of the original function has not been covered and it becomes necessary to create a delta (δ) function to complete the representation. This delta function is entirely non-symmetric and represents the "noise" that exists in the original

function; noise that cannot be accounted for by pulling symmetries. Since it does occur in this manner we cannot expect it to recombine with the symmetries during any later processing unless the final representation actually marries symmetries with some other properties. Because of this, the delta function can be thought of as an "essential term" in the same sense as used in several of the reduction schemes considered earlier [4, 6] ; and as such will be identifiable as a single element in the final equations and hardware.

Symmetries in f e d c b a	δ Function (noise)	Symmetric Function	Original Function
0 0 0 0 0 0	0	0	0
0 0 0 0 0 0	0	0	0
0 0 0 0 0 0	0	0	0
0 0 0 0 0 0	0	0	0
0 0 0 0 0 0	0	0	0
C 0 8 0 F F	0	F	F
0 0 1 1 0 0	0	1	1
C 0 0 8 0 C	0	C	C
3 1 2 0 0 3	0	3	3
0 0 0 0 0 3	0	3	3
3 3 0 2 0 3	0	3	3
0 4 4 4 6 0	0	6	6
0 2 0 2 0 0	0	2	2
0 C 2 4 9 C	0	D	D
0 0 4 0 0 0	0	2	2
0 8 0 0 0 0	2	8	A
8 8 6 6 8 14	1	21	22 bit count

Figure 8

If we consider the matrix of symmetries along with its delta function, we observe that the total bit count of 51 greatly exceeds the number, 22, required to represent the function. The reason is apparent when we look back and note that some elements of the function are redundant in the sense that they are used several times in forming

the symmetries. This redundancy is not necessarily wasteful since some of these excess bits will disappear during later processing and others becomes useful when trying to express the network in a new form.

Since partitioning into the present sub-functions was accomplished using the original function as a base, it follows that any addition, deletion or change in the columns which does not ruin our ability to recover the function is allowable. Thus we are able to further extend the partitioning process to the columns of our matrix and reduce the function into still simpler sub-functions. The criterion we use will be to partition the columns of symmetries into a set of "continuously foldable" functions. These are functions which will be defined as follows: A function is continuously foldable if beginning with the highest order symmetry and folding upward along the lines of symmetry causes all of the bits in the function to superimpose upon whas was initially the uppermost bit. The reason for this breakdown becomes more apparent after a method has been developed for compressing the data contained in the new matrix. Compression is necessary to permit manual or small memory computer manipulation of this data. It should be evident that further breakdown to the "continuously foldable" limit in no way destroys the symmetric properties. At most, some additional redundancies may be picked up which will only be removed in later processing.

For an example of the process consider the "f" symmetry of Fig. 8. The highest order symmetry is in "f". After folding the lower half of the function on to the upper half we have remaining:

0
0
0
0
0
C
0
C

The only other symmetry is in "a", and making these folds does not result in all bits being superimposed on the uppermost bit. Therefore the column which gives the degree to which the original function was "symmetric in "f" is not a continuously foldable function. By inspection the column is easily split into two functions which are continuously foldable and symmetric in "f". The first contains all zeros along with the uppermost "C" and the lower "3". The second function uses the remaining two hexadecimal elements.

This process of partitioning into continuously foldable functions could be easily mechanized by applying the following rules to a matrix of symmetries such as that given in Fig. 8:

(1) Initially subdivide the columns into a new set of columns each containing only "one type" of hexadecimal element. Since we are treating symmetries, "one type" includes any particular element plus the element which is obtained by folding the bits of the initial function. Thus "1" and "8" or "3" and "C" are each considered one type. In making this subdivision it should be remembered that the hexadecimal elements are only a form of notation and we must base our division on the underlying bits. For example, in column "e" of Fig. 3 the "3" contains a "2" and the "C" contains a "4", thereby allowing us to subdivide into a function containing two 2's and two 4's. This is shown by

the third column of Fig. 8.

(2) Test each of the new functions created by step one to determine whether or not the new functions are continuously foldable. Subdivide those which are not into a set of functions which still exhibit the same order symmetry but are now continuously foldable.

(3) Those functions which are completely asymmetric are the essential terms previously mentioned and remain isolated in their own columns.

Now applying these rules to the symmetries of Fig. 8 gives the new matrix of continuously foldable functions, Fig. 9.

$$
\begin{array}{ccccccccccccccccccc}
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
C&0&0&0&0&0&8&0&0&0&0&F&0&0&F&0&0&0&0\\
0&0&0&0&0&0&1&0&0&1&0&0&0&0&0&0&0&0&0\\
0&C&0&0&0&0&0&0&0&8&0&0&0&0&0&C&0&0&0\\
0&3&0&0&0&1&0&2&0&0&0&0&0&0&0&3&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&3&0\\
3&0&2&3&1&0&0&0&0&0&2&0&0&0&0&0&3&0&0\\
0&0&4&0&0&0&0&4&0&0&4&0&6&0&0&0&0&0&0\\
0&0&2&0&0&0&0&0&0&0&2&0&0&0&0&0&0&0&0\\
0&0&4&C&8&0&0&0&4&0&4&0&0&9&0&0&C&0&0\\
0&0&0&0&0&0&0&0&2&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&8&0&0&0&0&0&0&0&0&0&0&0&0&2\\
\mathrm{f}&&&\mathrm{e}&&&&\mathrm{d}&&\mathrm{c}&&&\mathrm{b}&&&&\mathrm{a}&&\uparrow
\end{array}
$$

Noise (δ) function

Figure 9

This new matrix carries with it all of the information of Fig. 8. The brackets indicate the symmetries of Fig. 8 which yielded these sets of continuously foldable functions.

7. Class system for matrix representation

Before the available information which is now contained in the matrix of continuously foldable functions plus its noise term can be processed efficiently, we must specify a more compact method of expression. It is immediately apparent that the individual columns of the matrix contain only one "type" of hexadecimal element; in fact the last partitioning performed placed the data in this configuration. Since the type of these elements (1 or 8, 3 or C, etc.) is the deciding factor for "a" and "b" symmetries, it is reasonable to suspect that they should also play an important role in determining whether the input variables "a" or "b" occur in the final logic. We shall see that this is the case and for this reason they are the basic class (Class I) of our class system.

Next dividing the columns into groups of four hexadecimal elements we notice that elements within these groups of four take on a pattern which also lends itself to the hexadecimal notation. Finally, we can consider the four groups of four, and imagining that a one bit is on the field where the hexadecimal elements other than zero occur, we can again express the layout hexadecimally. Fig. 10 shows the matrix with these groupings designated as Class I, II, and III.

The Class I grouping represents the type of hexadecimal element and is always expressed as the element which appears first in the given column. The Class II grouping is the hexadecimal layout of Class I elements. The Class III grouping is a hexadecimal layout of Class II elements. This method can be continued to any number of variables. For each additional pair of input variables the

number of classes increases by one. In the event of an odd number of variables the number of hexadecimal elements possible in the highest order Class becomes limited, and in the manipulations which follow the Class would be automatically considered symmetric in the highest order variable.

$$
\text{Class III}\left\{
\begin{array}{c}
\text{Class I}\\
\downarrow\\
\begin{array}{ccccccccccccccccccc}
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
\hline
\end{array}\\
\left.\begin{array}{ccccccccccccccccccc}
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
C&0&0&0&0&0&8&0&0&0&0&F&0&0&F&0&0&0&0\\
0&0&0&0&0&0&1&0&0&1&0&0&0&0&0&0&0&0&0\\
0&C&0&0&0&0&0&0&0&8&0&0&0&0&0&C&0&0&0
\end{array}\right\}\text{Class II}\\
\begin{array}{ccccccccccccccccccc}
\hline
0&3&0&0&0&1&0&2&0&0&0&0&0&0&0&3&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&3&0\\
3&0&2&3&1&0&0&0&0&0&2&0&0&0&0&0&3&0&0\\
0&0&4&0&0&0&0&4&0&0&4&0&6&0&0&0&0&0&0\\
\hline
0&0&2&0&0&0&0&0&0&0&2&0&0&0&0&0&0&0&0\\
0&0&4&C&8&0&0&0&4&0&4&0&0&9&0&0&C&0&0\\
0&0&0&0&0&0&0&0&2&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&8&0&0&0&0&0&0&0&0&0&0&0&0&2
\end{array}
\end{array}\right.
$$

Figure 10

With this new class notation the matrix of continuously foldable functions is expressable as given in Fig. 11. This table will be referred to as the Hexadecimal Array

Class I	C	C	2	3	1	1	8	2	4	1	2	F	6	9	F	C	3	3	2
Class II	4	1	3	2	2	8	6	9	6	3	3	4	1	4	4	1	2	4	1
Class III	6	6	3	3	3	3	4	2	1	4	3	4	2	1	4	6	3	2	1

Figure 11

As an illustration of the procedures followed in developing the alphanumeric terms of the Hexadecimal Array consider the third column of Fig. 10. This column requires more than two elements since it is symmetric in "e" and symmetric in "c". In the actual writing of terms it is more convenient to begin by considering the highest class first even though this requires writing the terms from bottom to top. By following this procedure it is unnecessary to carry over information about one class while working with a second class. First looking at the Class III layout we observe that the elements cover the quartered field in a "3" configuration. Consequently, the bottom element of the alphanumeric term is a "3". In the Class II layout, we scan the column from the top stopping in the first quarter which contains hexadecimal elements. The configuration of these elements uniquely defines the element to be recorded above the "3". Since, in the example taken, this amounts to a "2" and "4" in a "3" configuration, the next element is a "3". Finally the basic "type" element of the column becomes the Class I part of the term. Thus in a second top to bottom scan of the column, the first non-zero element encountered becomes the uppermost element of the Hexadecimal Array. Hence the complete alphanumeric term that has been constructed is recorded in the third column of Fig. 11 as $\begin{matrix}2\\3\\3\end{matrix}$. This expression is unique in that the original column can be readily re-constructed from this notation.

Next we notice that when the Hexadecimal Array is expressed in binary we have reduced the number of bits required to carry the information of each column from 64 to 12. This represents a

saving of considerable importance when we reach the stage in which these systems might be programmed for digital computers or represented directly by logical networks. Of even greater importance to the present investigation is the fact that we have now introduced a notation which in itself suggests that there must be means for processing our data to obtain some reductions. Further, it is possible to develop rules which would transform the symmetries of Fig. 11 by a two step operation into congruencies.

Since all of the above operations which we have either described or mentioned could be represented by mechanical steps they are readily adaptable for computer programming. However this must of necessity be left for future investigation. The scope of the present investigation cannot be extended to include the efficient marrying of the two methods, symmetries and congruencies or to programming. It should suffice to say that work toward this goal is being continued.

8. Reduction of symmetric functions

The initial function has now been completely partitioned into a form, the matrix of continuously foldable functions plus a noise term, which suggests that some form of symmetric hardware along with "or" circuits could be used to express the network. This is indeed true, but as the function stands there are nineteen terms which seems to imply a need for at least that number of hardware components. Since this would be unrealistic for an initial function which started out with only 22 bits, we must develop schemes for obtaining some savings. The reductions in the cases which follow do not imply a minimal form of logic or an optimum use of hardware, but have been developed and presented as a guide to reductions in general.

The most obvious reduction has been evident since the matrix of Fig. 9 was calculated, and can be effected simply by removing the duplicity of terms or sub-functions. These have been carried along naturally to insure that any final system be directly mechanizable. For this reason, our development is following through in a fashion similar to the steps which would be required of a machine.

```
C C 2 3 1 1 8 2 4 1 F 6 9 2 3
4 1 3 2 2 8 6 9 6 3 4 1 4 1 4
6 6 3 3 3 3 4 2 1 4 4 2 1 1 2
```

Figure 12

Fig. 12 is the new array derived directly from the Hexadecimal Array of Fig. 11. The duplicated terms have been removed by scanning from left to right, dropping all second occurrences.

It is interesting to note that a scan from right to left produces the same set of alphanumeric terms but orders the terms in a new sequence. Whether or not the sequence of ordering the ensemble will effect future developments has not been decided. This may be answered when additional research determines the effects of scanning and sorting such functions or systems of functions in accordance with their inherent statistical and other properties. Such properties can be the frequency of a particular input variable in an intermediate set of developing equations, the tendency of the function or system toward a "type" of hardware, etc.

At least one method has been developed which enables us to remove some of the redundancies contained in the new array. However, we should not always assume that it is advisable immediately to remove redundancies. Their worth is obvious from having considered applications where redundant "forbidden states" lead to greatly simplified logic. M. Phister, in his book, "Design of Digital Computers", [15] gives a good example of the procedures used to treat these redundancies when operating with normal Boolean expressions. Although we are not dealing with the same type redundancies, they are nonetheless important whenever methods for combining terms of equations or hardware are being considered. Unfortunately, efforts to date have been only partially successful in being able to bring about any significant combinations of terms into a truly reduced[1] form. The methods involve a somewhat dubious processing of the basic bit

---

[1] "Reduced" - again from the viewpoint of overall system and state-of-art hardware.

data and as yet have not been standardized. For this reason, no effort will be made to include such techniques in the present paper.

The method for removing redundancies which has been formulated is offered mainly to be used as a technique to reduce further the dimensions of the latest representation of the function, rather than as a means of insuring any form of reduced logical expression. We might best describe the method in a series of steps.

Step One. Each of the alphanumeric terms of the Hexadecimal Array must be further sub-divided into its simplest elements. To do this we consider the elements of the individual terms from top to bottom in the following manner.

(a) No split is possible if each of the hexadecimal characters of the term is an 8, 4, 2 or 1. At this point only noise terms will have this characteristic since all other terms contain at least one symmetry. A symmetric subfunction implies an even number of canonical terms.

(b) If the first element is a hexadecimal character composed of two or more bits, the initial division will be into this number of new terms. Each new term will have as its first element one of the hexadecimal characters 8, 4, 2 or 1 such that when we perform a logical addition of first characters the upper element of the original term is returned. The remaining elements of the new terms will be identical with similarly placed elements of the parent term. See Fig. 13(a).

(c) In cases where the first element is a hexadecimal character containing only one digit and at least one of the remaining elements contains more than one bit, the first

element makes a two-way split into the two elements of its "type", i.e. an "8" goes into an "8" and a "1"; the original element appearing first. The remaining elements are treated in similar fashion, recording the two elements of the "type" appearing in the same position of the parent terms as was done above, until coming to a multi-bit hexadecimal character. When this character is encountered it is sub-divided as were the elements of step 1 (b). However, in the present situation the ordering of new elements must be from the sequences 8, 4, 2, 1. Ordering as used here implies two requirements. First, the characters must be selected from the set 8, 4, 2, 1; and secondly, the characters must be sequenced in the same order. Thus, the characters "8" and "1" must appear in this order. As before, once a split of a hexadecimal character has occurred the remaining terms are merely repeated as they occurred in the original term.

(d) Following the rules of (b) and (c) we will have sub-divided the original term into either two or four new terms. Each of these new terms is now treated as a new original term and sub-division in accordance with (a), (b) and (c) continues. This process is repeated until the final terms are composed exclusively of hexadecimal characters from the set 8, 4, 2, and 1. The final number of terms can be predicted at the start by forming the arithmetic product of the numbers of one bits used to form each of the hexadecimal characters of the term.

For example, the elements of the term $\begin{matrix}C\\4\\6\end{matrix}$ are "C", "4" and "6". "C" requires two one bits, "4" one one bit and "6" two one bits. Therefore, two times one times two equals four sub-divided terms.

By now it should be apparent that a split into three terms is not possible since the "folding" process has a binary nature in any symmetric system. In fact, because of symmetries the only hexadecimal characters allowed are 1, 2, 3, 4, 6, 8, 9, C and F. Three-bit terms and "5" and "A" are not symmetric.

In Figs. 13(a) and 13(b) the two stages of the breakdown for $\begin{matrix}C\\4\\6\end{matrix}$ are shown. Fig. 13(c) gives a more complex example, $\begin{matrix}6\\6\\6\end{matrix}$. In this latter case, three stages of breakdown are necessary.

$$\begin{matrix}C\\4\\6\end{matrix}\rightarrow\begin{matrix}8\\4\\6\end{matrix}+\begin{matrix}4\\4\\6\end{matrix}$$

(a)

$$\begin{matrix}C\\4\\6\end{matrix}\rightarrow\begin{matrix}8\\4\\6\end{matrix}+\begin{matrix}4\\4\\6\end{matrix}\longrightarrow\begin{matrix}8\\4\\4\end{matrix}+\begin{matrix}1\\2\\2\end{matrix}+\begin{matrix}4\\4\\4\end{matrix}+\begin{matrix}2\\2\\2\end{matrix}$$

(b)

$$\begin{matrix}6\\6\\6\end{matrix}\rightarrow\begin{matrix}4\\6\\6\end{matrix}+\begin{matrix}2\\6\\6\end{matrix}\rightarrow\begin{matrix}4\\4\\6\end{matrix}+\begin{matrix}2\\2\\6\end{matrix}+\begin{matrix}2\\4\\6\end{matrix}+\begin{matrix}4\\2\\6\end{matrix}\rightarrow\begin{matrix}4\\4\\4\end{matrix}+\begin{matrix}2\\2\\2\end{matrix}+\begin{matrix}2\\2\\4\end{matrix}+\begin{matrix}4\\4\\2\end{matrix}+\begin{matrix}2\\4\\4\end{matrix}+\begin{matrix}4\\2\\2\end{matrix}+\begin{matrix}4\\2\\4\end{matrix}+\begin{matrix}2\\4\\2\end{matrix}$$

(c)

Figure 13

Step Two. Next we form the table shown in Fig. 14. This is done by first recording the Hexadecimal Array of Fig. 12 to form the upper row of the table and then listing under each of these terms the sub-divisions determined in Step One. The ordering of the sub-divisions is important. They will be listed in the order as derived in Fig. 13.

Step Three. This consists of a scan to remove the actual redundant terms without which the original function still has complete cover. As was mentioned previously, the procedure for conducting this scan is as yet arbitrary. No truly optimum method can be specified until the overall techniques can be furthered to include a minimal network concept. Several types of scans have been made.

(a) A simple right to left scan. A term appearing on the top row (terms originating from the array of Fig. 12) must be completely covered by other terms from the same set before any reduction is permitted. A sub-division of each term is picked up and held until all sub-divisions to the left have been scanned. If complete coincidence exists, we continue to examine the other sub-divisions of the term being considered until either a sub-division is held that is not duplicated or it has been determined that all sub-divisions are repeated. In the former case we move on to check the next term to the left following the same test pattern; otherwise we strike the complete column as redundant. In scans made after the extreme right column has been checked, the sub-division held is considered coincident even though it exists only in a column to the right which has been previously examined and found to lack complete redundance. Scanning continues until each of the terms of the upper row have been examined. Fig. 14 is an example using this form of scan.

(b) A left to right scan. This scanning process is identical to (a) above except the terms are examined beginning at the

left end of the upper row and moving to the left.

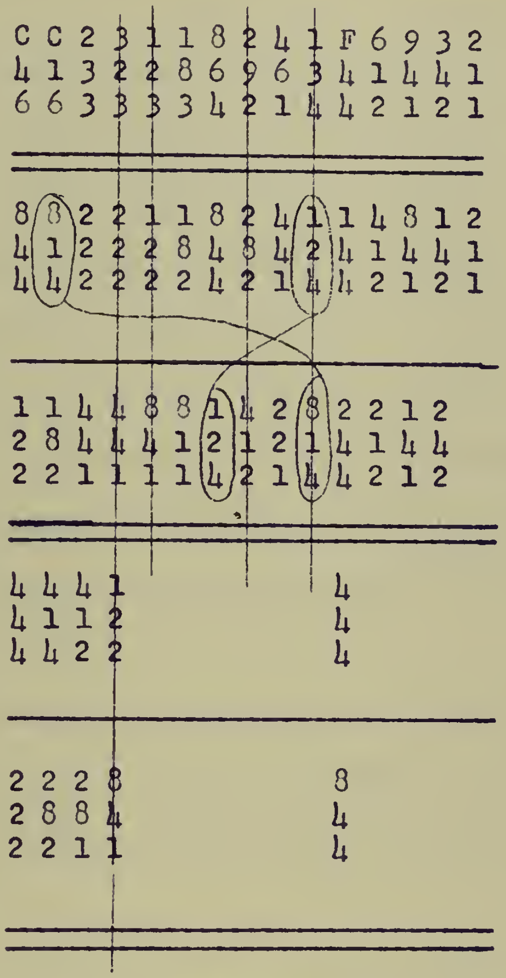


Figure 14

(c) Scans which leave residue. Again the actual scanning process is the same in most respects to those above and can be started at any point, either end or at some internal point. The difference rests with the manner in which we strike redundancies. We permit a further reduction of the initial terms into a residue consisting of one or more combinable subdivisions. Such a condition exists if during the scan coincidence exists between all but one of the

subdivisions of the term being considered. Combination is possible if two, four, eight, or sixteen terms remain which are completely contained by pairs of double lines in the table of Fig. 14. An Additional requirement is that this group form a natural segment of the column if the column were divided into halves, quarters, eights or higher order divisions. Where the number of input variables is increased the sizes of these groups may be larger. The purpose of the division in the above manner was to insure that the terms would readily recombine into a single reduced term by simply applying rules which are the inverse of those stated in Step One.

The encircled terms of Fig. 14 is an example which shows the striking of $\begin{matrix}1\\3\\4\end{matrix}$ from the hexadecimal array. The three additional vertical lines cross through the remaining redundant terms.

```
C C 2 1 8 4 F 6 9 3 2
4 1 3 8 6 6 4 1 4 4 1
6 6 3 3 4 1 4 2 1 2 1
```

(a)

```
C 2 1 2 4 1 F 6 9 3 2
1 3 8 9 6 3 4 1 4 4 1
6 3 3 2 1 4 4 2 1 2 1
```

(b)

```
C C 2 1 1 4 3 6 9 3 2
4 1 8 8 2 6 4 1 4 4 1
6 6 1 3 4 1 4 2 1 2 1
```

(c)

Figure 15

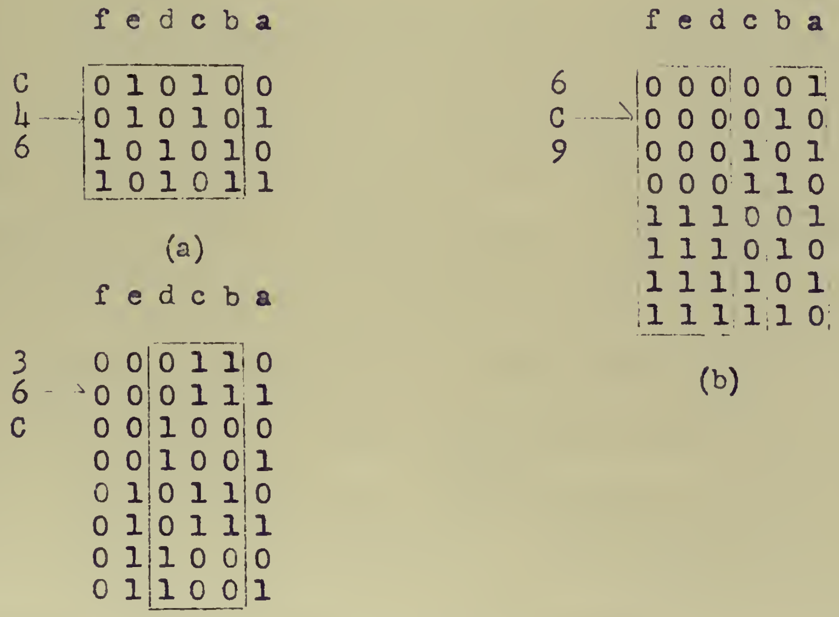
Step Four. Finally all of the terms remaining after the deletion or deletion-and-reduction processes of Step Three are collected to form the new arrays of Fig. 15. Although these arrays still represent the original function exactly, all possible solutions are no longer implicit as a result of the above reductions. For example, the array of Fig. 15(a) was obtained by a simple right to left scan and will produce a different final solution when hardware is postulated than the arrays of Figs. 15(b) and 15(c) which were arrived at by a simple left to right and residue type scans respectively.

9. Logical representation of symmetries.

Starting from the basic concepts of symmetries we have carried through a partitioning process to develop the Hexadecimal Array of Fig. 11. Methods were formulated to show that it is possible to reduce the dimensions of the array and remove some of its inherent redundancies without destroying the symmetric properties of the terms. However, looking at the results, we are aware that the information as it stands is in a language unfamiliar to those accustomed to treating logical expressions in Boolean format.

It is not possible to express each term of Fig. 15 as a single logical product term of the type produced by Class I reduction techniques. The nearest thing to the language of our Hexadecimal Array was produced when partitioning was into congruences [18], but even there the final logical network was represented by Boolean equations. In the present case, a symmetric or reflecting environment has been forced on us by the processing itself which provides a natural specification for discrete, symmetric hardware. Before introducing the means by which we can go directly from the Hexadecimal Array to this symmetric form of hardware, we must look at a breakdown of the terms themselves and derive a basis for writing the new non-Boolean expressions. We should note that, although reference to non-Boolean logical expressions and equations will continue, there is a correspondence which would permit us to convert the final equations into Boolean form. However, there is no advantage at present for making such a conversion.

Fig. 16 gives the bitwise subdivision of three hexadecimal terms into their symmetric elements. $\begin{matrix}C\\4\\6\end{matrix}$ should be recognized as the first term of Fig. 15 (a) while $\begin{matrix}6\\C\\9\end{matrix}$ and $\begin{matrix}3\\6\\C\end{matrix}$ are two arbitrary terms used here to illustrate the general properties of hexadecimal terms. In each case the number of subdivisions can be predicted by using the product method given in section eight under step 1 (d). Applying these rules, we have that $\begin{matrix}6\\C\\9\end{matrix}$ and $\begin{matrix}3\\6\\C\end{matrix}$ each yield eight basic canonical terms and $\begin{matrix}C\\4\\6\end{matrix}$ yields four. Rules could be stated for writing these canonical terms directly from the hexadecimal terms but for present purposes it is sufficient to realize that they can be obtained by using the basic N-bit counter[1]. This is done by expressing the hexadecimal terms as subfunctions of ones and zeros alongside the counter and removing the canonical set of minterms indicated by the one bits.



(c)

Figure 16

---

[1] In the present example N = 6

A study of a large number of subdivided terms such as those given by Fig. 16 leads to the following observations:

(a) Groups of bits always occur which are composed of only one particular expression of ones and zeros and its complement for the length of the columns. In Fig. 16(a) the expression is 01010 and the complement 10101; in Fig. 16(b) there are two such groups the expression and its complement in the first are 000 and 111 respectively and in the second 01 and 10; and in Fig. 16.(c) the expression is 011. The number of bits in these expressions can range from two to the total number of variables used in the table.

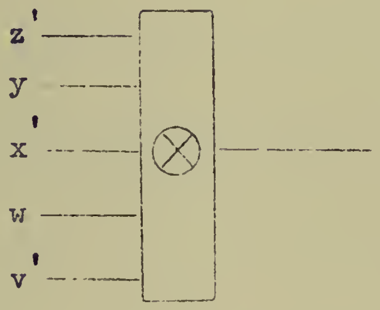
(b) In cases were the subfunctions are asymmetric in higher order variables, columns at the extreme left of the bit configuration do not change over the range of the subfunction, i.e., they remain as a one or a zero for the length of the column as in column "f" of Fig. 16(c).

(c) Single columns which occur other than as in (b) above are not needed in expressing the subfunction. They merely turn up as a zero and a one for each of the other configurations of the table. Examples of this are the "a" column of Fig. 16(a), the "c" column of Fig. 16(b), and the "e" and "a" columns of Fig. 16(c). Such occurrences are similar to what takes place in Class I methods such as that of Quine [6] where the single variable will disappear and the terms combine when the variable appears in both its true and false states along with an unchanging set of co-variables. This is illustrated by the variable x in the expression:

$$xy'z + x'y'z = y'z$$

The above observations point clearly to some form of complementing device capable of handling an input of "n" independent variables, and providing outputs when these variables are voltage-wise either all high or all low. This represents an extension of the general "exclusive or" circuits which have been engineered to handle only two variables. Since the more general complementing devices provide outputs for two conditions of the input variables, they will herein be called "bigates". The transforming property of this device is illustrated in Fig. 17.

This symbol $\otimes$ will be used in logical schematics to represent



$$v'wx'yz' + vw'xy'z = \underline{v'wx'yz'}$$

Figure 17

this new logical component. To permit the writing of the non-Boolean equations previously mentioned, expressions such as the output function of Fig. 17 will be written as one term underscored. Whenever these underscored terms occur in logical equations they imply the use of bigates in final hardware configurations.

The bigate is not envisioned as a mere combination of existing circuit components, but rather as a special transistor-like element

capable of handling multiple inputs and at least one output. These circuit elements could be used in combinations along with standard "and" gates to represent any of the symmetric subfunctions so far developed. For example, the three hexadecimally represented terms which were subdivided in Fig. 16 lead to the hardware configurations of Figs. 18(a), 18(b), and 18(c). Note that the configuration of Fig. 18(b) gives an output under four sets of conditions. This results from the fact that the expression $\underline{ab'}\ \underline{d'e'f'}$ contains two underscored groups which are present in either of two states. By induction N bigates would imply N factorial (N!) combinations each leading to an output.

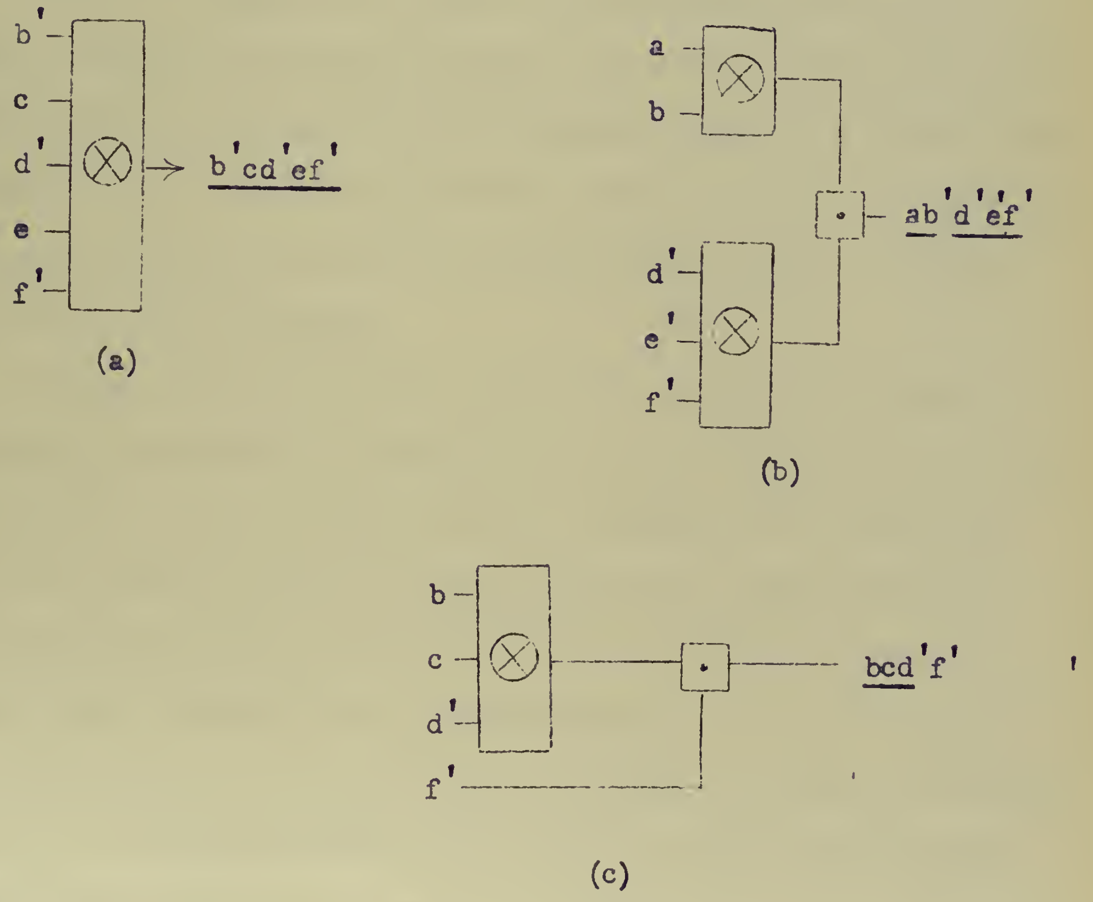


Figure 18

In Fig. 18(c) the "f" input does not go through a complementing device. Instead this term must be present in its false state before it is possible to obtain an output from our device. Terms with these properties (see observation (b) above) might be used to adjust the "bias" levels of the bigate variables. The schematics used for illustrative purposes have pictured the bigate as a grouping of elements which could be conventional or new forms of hardware. However, to achieve optimum use of configurations based on the symmetric properties of a function, it would be advantageous to have single elements capable of being biased and performing the complementing function simultaneously.

Using the bigate as the basic building block and the ideas just developed, we are able to write a set of rules which enable us to go directly from terms of the Hexadecimal Array to a logical representation of the initial function. We should mention here that any expressions will still be in an uncombined form which, by previous criteria, are non-minimal.

Before developing specific rules for writing these logical expressions, consider the table given in Fig. 19. This table was constructed for use with the illustrative example being followed but can be adapted and used with functions having a greater number of variables by simply expanding that portion of the table above the solid line. The lower case letters appearing in the two columns immediately above the two bit counter represent the system variables. Those in the leftmost column are referred to as the higher order variables, while those to the right are the lower order variables.

In this table we have listed alongside the two variable counter the nine hexadecimal characters which can occur in descriptions of symmetric subfunctions. This enables us to select the variables

ba	1	2	3	4	6	8	9	C	F
dc	1	2	3	4	6	8	9	C	F
fe	1	2	3	4	6	8	9	C	F .
00	0	0	0	0	0	1	1	1	1
01	0	0	0	1	1	0	0	1	1
10	0	1	1	0	1	0	0	0	1
11	1	0	1	0	0	0	1	0	1

Figure 19

controlled by a given hexadecimal element and also indicates in which of its states (true or false) the variable should appear. A study of the symmetric properties of these characters shows that the following procedures can be applied:

(1) The variables ordered by subfunction elements are located from the table by moving left on the line corresponding to the location of the elements in the Hexadecimal Array.

(2) The single bit characters 1, 2, 4 and 8 order the two variables in the states indicated by the counter elements opposite this bit and in the indicated state, e.g., if "4" occurs as the bottom element of the hexadecimal term it orders the variables f' and e.

(3) For the bi-bit characters 3, 6 and 9 and C we consider only the uppermost bit when we select the variables and their states. The symmetries automatically handle the additional bit contained in each of these elements. As an example, the character "3" as the

bottom element of our six variable function orders the variables f and e'.

(4) The character F always orders a "one" along with a lower order variable in its false state. The one results from the fact that there is complete symmetry in both variables while the lower order variable comes from the upper bit of the F in a manner similar to (3) above. Thus if "F" occurs as the bottom element of an example it also orders the variable e'. The "one" is meaningless and adds nothing to the expression.

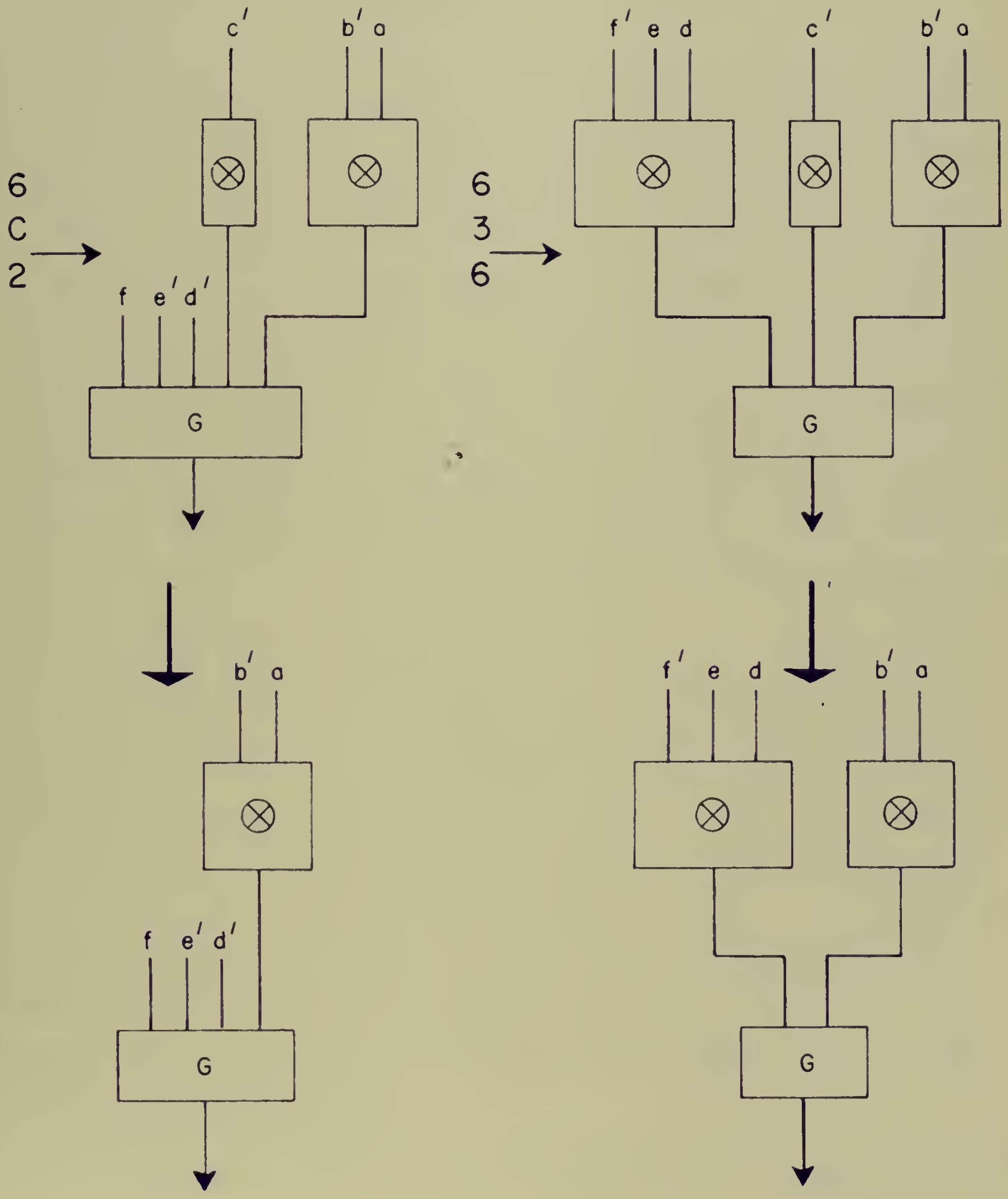
As we noted earlier, bias variables, if they occur, begin with the highest order variable of the basic counter; in our case f. The number of such variables which can occur ranges from zero to N, where N is the total number of variables, and can be predicted by scanning the variables of the subfunction to determine where the first symmetry occurs. This scan begins with the highest order variable, which amounts to looking at the bottom element of a hexadecimal term, and continues toward the lower order variables. Thus if a subfunction has symmetry in the highest order variable there is no bias, but if the first symmetry occurs later in the scan all variables already scanned become bias variables. This set of variables represents the total bias of the expression. After removing this bias the remaining variables either form inputs to one or more bigates or are not needed.

The unidirectional scan to remove the bias variables necessarily begins with the bottom element of the subfunction and progresses upward. This is equivalent to searching from the highest to lowest order variable.

Using the table of Fig. 19 and the procedure stated in conjunction with this table we can now write the bias variables directly from the hexadecimal terms. First we might note as an example that the hexadecimal characters 6, 9 and F are each symmetric in the higher order of the two variables; in this case f, d and b. If any of these terms appear as the bottom element of the subfunction, bias variables will not be present. On the other hand, the characters 3 and C are asymmetric in the higher order of the two variables but symmetric in the lower order variable. Hence, if these characters appear as the bottom element, single variable bias is generated. The remaining single bit hexadecimal characters; 1, 2, 4 and 8; are completely asymmetric and consequently each generate the two variables of bias indicated in Fig. 19. If any of these four characters occur as the bottom element of the subfunction the scan must continue to the next element. Thus we assemble bias variables until the scan encounters the first symmetry. In the example of Fig. 20(a) this occurs in the middle element, C, which is symmetric in the lower order variable. Since there are no symmetries in the higher order variables all of these variables together form the bias for our logical expression. An additional example is shown in Fig. 20(b). For a complete representation of the initial function see Fig. 21.

After removing the bias variables we can apply the following rules to develop the remainder of the logical expression and hence hardware configurations. These rules and what has been said previously apply for any number of variables.

(1) As soon as the first symmetry is encountered, we close off the bias and begin writing the variables ordered from Fig. 19 as the



(a)

(b)

FIGURE 20

inputs to the first bigate. For example, if the bottom element of a six variable subfunction is "C", f' is bias and e' is an input to the bigate.

(2) The remaining elements are included as inputs to the same bigate until a 3, C, 6, 9 or F appears. The highest order variable of either a 3 or C, or one in the case of F, is recorded as the input to the bigate. These three characters each propagate the false state of the lower order variable into the next bigate. This is best illustrated by a subfunction such as $\begin{matrix}4\\C\\C\end{matrix}$. The C as the bottom element gives a bias of f' while e' is propagated into the first bigate. The second C scanning upward introduces an additional input, d', into this bigate and propagates a c' on to the next bigate. The 4 adds two more inputs, a and b', to the second bigate.

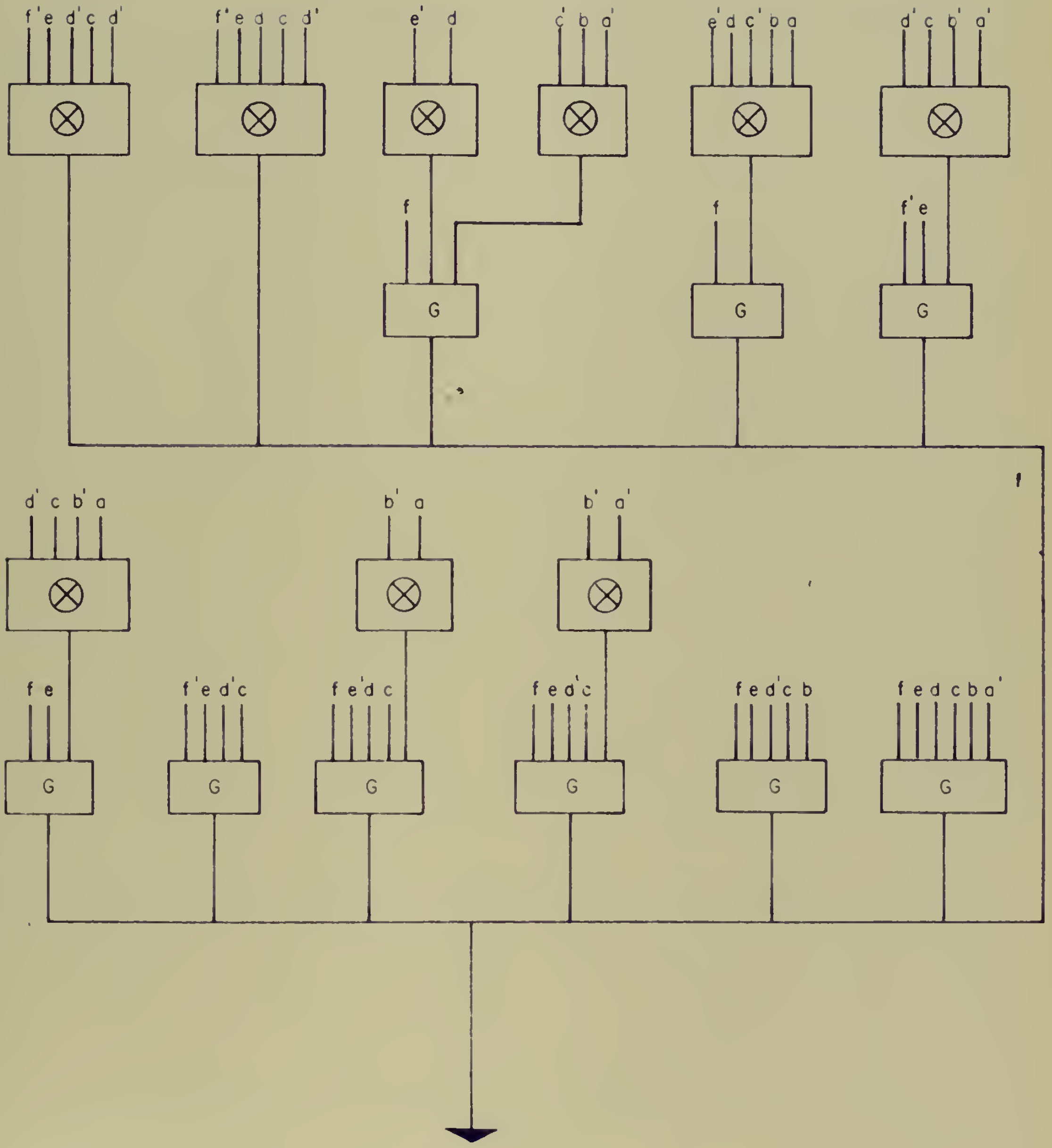
(3) Bigates are not required if they receive only one input since this implies that this variable is always present in both its states. When this occurs the bigate and the variable are deleted.

(4) A 6 or 9 breaks the above pattern in that whenever they occur we must automatically disassociate ourselves from the bigate begin considered and begin loading the variables ordered from the table of Fig. 19 by the 6 or 9 into the next bigate. Fig. 20 (b) illustrates this property.

In Fig. 21 the letter "G" is used to indicate conventional "and" gates. These gates would be eliminated in those cases where they can be associated with a bigate if the final components can be developed with this property.

Before concluding it is interesting to note that even though the general bigate is not available as a single hardware component there

CC2184F6932
41386641441
66334142121



$$F = \underline{f'ed'c\,d'} + \underline{f'edcb'} + f(\underline{e'd})(\underline{c'ba'}) + f\underline{e'd'cba} + f'e\underline{dc'b'a'}$$

$$+\ fe\underline{d'cb'a} + f'ed'c + fe'dc\underline{b'a} + fed'c\underline{b'a'} + fe'd'cb + fedcba'$$

FIGURE 21

is at least one circuit which can be used when there are only two inputs. This is the generalized "exclusive or" circuit described in Appendix B. By proper arrangement of inputs (three in each case) the four bigates which operate on the inputs xy, xy', $x'y$ or $x'y'$ can be instrumented.

10. Combinatorial properties of symmetric subfunctions.

Fig. 21 displayed the normal results of processing a given function mechanically along the lines of its symmetric partitions. It is roughly analogous to a two-level gating network obtained through Boolean methods. At this point we could entertain the prospects of operating in the time domain in lieu of, or along with, parallel processing. This was suggested when we observed that processing was performed by unidirectional scans of the occurring signals. Processing is always from most to least significant bits and as we progress there is no requirement to remember what was done with previous bits. However, the parallel network concept is more in keeping with the aim of this investigation where we are trying to accomplish a task in one gating period with the simplest hardware combinations available.

One subject which should be considered at this point is the combinatorial properties inherent in our employment of bigates. We can illustrate the characteristics of the bigate along these lines using the simple example given in Fig. 22.

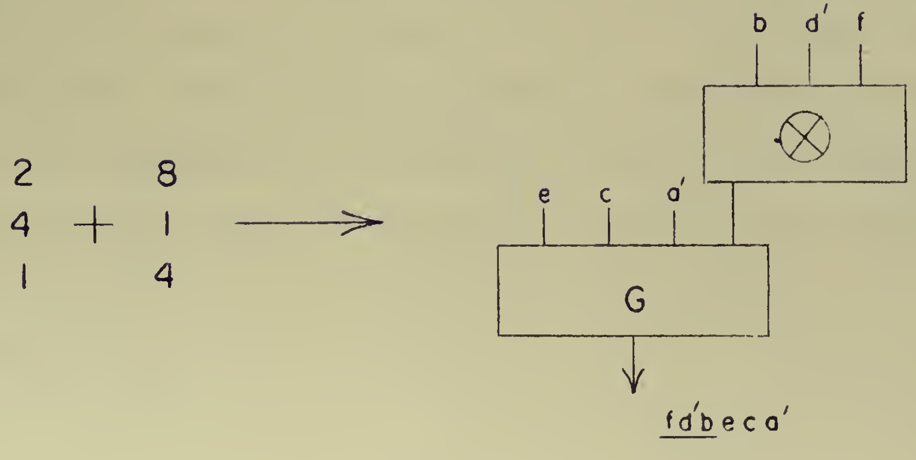


Figure 22

We postulate any two functions which must be expressed by employing all the input variables. These may be regarded as two wholly biased functions, or noise functions, and have no immediately obvious interrelationship, either in the domain of congruence or of symmetry. Their Boolean expression is fed'cba + f'edcb'a', which we can express in the hardware configuration of Fig. 22. Notice that the bias variables, eca', are no longer restricted to consecutive higher-order variables, but can now exist with breaks.

With the new combining technique available, we should now refer to an observation made earlier in Section 8. It was illustrated there that in casting out non-essential terms one might go so far as to cast out non-essential components of existing terms, carrying out what was labeled "scans which leave residue". It might be noted in the example shown in Fig. 15(c) that more than one wholly biased term was present. Clearly these can be swept together in pairs using the above technique.

We can provide a relatively comprehensive illustration, Fig. 23, of these principles by operating on the function partitioned in Fig. 15. Let us carry out the scan-which-leaves-residue operation exhaustively so that we might have a term by term breakdown. Terms which remain intact will not be altered. Terms which must yield some of their components to the elimination process will have their remaining parts split into noise components, or wholly biased parts. In this illustration we are approaching the level of listing each individual minterm.

C	C	4	4	2	8	3	8	1	2	2	1	3	2
4	1	4	1	8	4	4	1	2	2	1	4	4	1
6	6	1	2	1	1	4	1	4	1	2	1	2	1

Figure 23

We notice at once that $\begin{matrix}C\\4\\6\end{matrix}$ and $\begin{matrix}C\\1\\6\end{matrix}$ could combine into $\begin{matrix}C\\5\\6\end{matrix}$ were we to allow the non symmetric symbols A and 5. Here is the first hint that congruent partitioning can combine with symmetric partitioning advantageously, since A and 5 are characteristic of congruent partitioning. However, when we entertain the thought of combining the hexadecimal expressions for congruent and symmetrical partitioning we run into a host of semantic problems. The hexadecimal array in either system is not compatible with that of its correspondent. This is not unnatural when we consider that a property-wise perfect statement in one system is generally "noise" in the other.

There is a point of contact, however, in the terms which are all biased. When all the variables are required to define a term in the congruent scheme, the expression which appears is identical to the symmetrical expression for the same term. Thus $\begin{matrix}2\\1\\1\end{matrix}$ which is fedcba' in the symmetrical system has the same significance in the congruent system.

We can begin with this common point and evolve a method of expressing symmetrically derived terms having more randomly distributed biasing components. Up to this section we were limited to defining "bias" signals as the first asymmetricallsignals inherent in the hexadecimal terms when viewed from bottom to top. We can say that a signal which is not bias is a part of an "oscillator", and when scanning from bottom to top, independent oscillations appear in

what we can call "nth order" oscillations. Thus the hexadecimal term $\begin{matrix}2\\3\\3\end{matrix}$ when scanned upward has first the bias signal f then a first order oscillator e'd, and a second order oscillator c'ba', yielding the complete expanded set,

fe'dc'ba' + fed'c'ba' + fe'dcb'a + fed'cb'a

This is shown as the third term of Fig. 21. When treating the hexadecimal expressions in this manner, we preclude the existence of "mixing" among the types. Our unidirectional scan does not allow us to select a term with a "c" bias, for example, if any terms previous to it have been involved with an oscillator.

This situation has an affect on the recombination of terms which is clearly inhibitory. We require now the ability to perform two jobs, (1) to select terms which can combine advantageously, and (2) to express the effect of combining these terms. While neither of these tasks has been made to respond to a compact set of rules, an illustration using an unrefined procedure is provided. For ease of recognition, we will refer to terms in their basic precombined state. As we mentioned before, this puts us on a level corresponding to the minterm array. Experience with manipulation renders such extreme simplification unnecessary and computing machinery would process the data far more directly.

First we shall establish a basic method whereby combinable terms can be selected. "Combinable" here now means a more random distribution between oscillator and bias elements will be available. It should be pointed out that this illustration will not allow us to proceed beyond bias terms and first order oscillators, but this is sufficient to point the direction combinatorial methods being developed might take.

Consider an array of four hexadecimal terms having to do with a six variable input. We wish to bring these terms into a single expression if possible. We can do so if we arrange the terms so that they will satisfy the following requirements:

a. One row having the four separate elements 1, 2, 4 and 8

b. Any other row having two pairs of elements from the set 1, 2, 4 and 8.

c. The remaining row having all four elements fo the same character, again from the set 1, 2, 4 and 8.

Having selected hexadecimal arrays which fit the above description, we find that we have in reality suggested one set of operations common to all of them. That is, in a system having six input variables, we will use five of them always in the configuration given by Fig. 24. One variable will not appear.

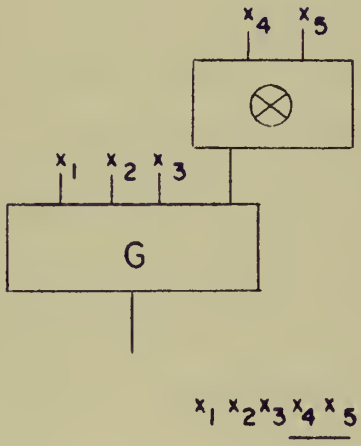


Figure 24

With reference to terms extracted from Fig. 23, we can then say we have the three sets:

$$\begin{array}{ccccc} 1\;2\;4\;8 & \qquad & 2\;2\;2\;2 & \qquad & 1\;2\;4\;8 \\ 4\;4\;4\;4 & & 1\;2\;4\;8 & & 4\;1\;4\;1 \\ 4\;4\;1\;1 & & 2\;1\;2\;1 & & 1\;1\;1\;1 \\ \\ (a) & & (b) & & (c) \end{array}$$

which are completely satisfied by our selection rules as reflected in the block diagram of Fig. 24 when the variable distribution is:

	x_1	x_2	x_3	x_4	x_5
(a)	e	d'	e	b	f'
(b)	a'	b	f	c	e'
(c)	c	e	f	a	d

Were we to specify another selection criterion for obtaining combinable sets from a subject function, we would be specifying another configuration of bigate and conventional "and" gate. We are confronted now with another class of data, i.e. all the applicable selection criteria which can be used. The creation and application of selection criteria cannot be an arbitrary process. It must proceed under the influece of two major considerations:

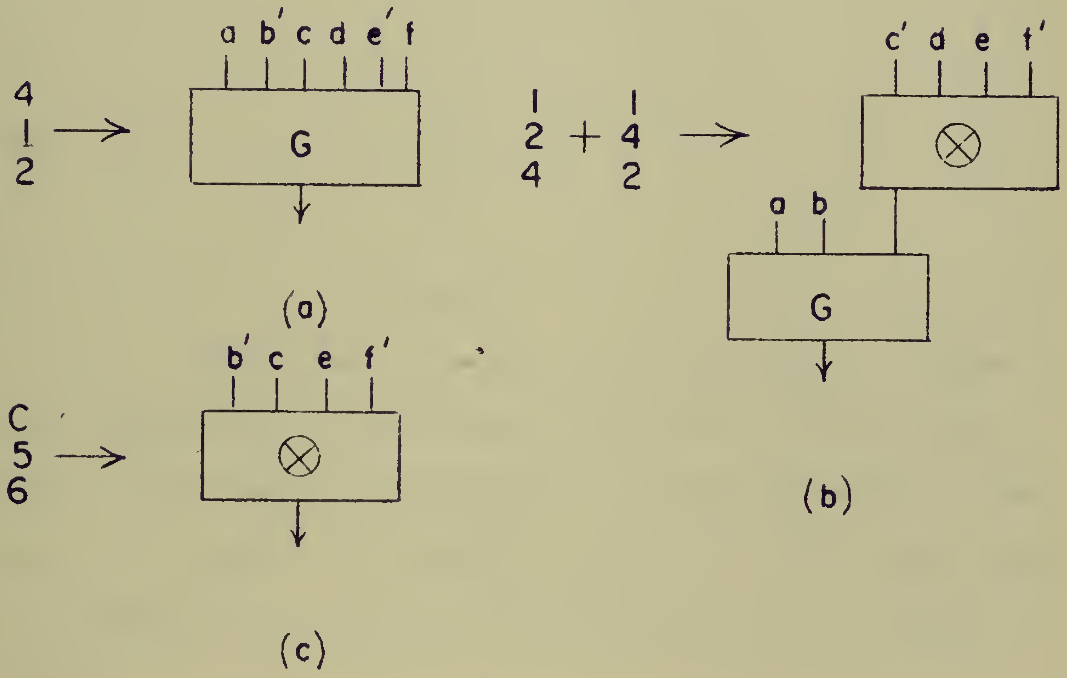
(1) The type and capacity of available circuit components, and

(2) The continuity over a system of functions.

Should the processing be contained in computing machinery, the first consideration would be entered as a constraining parameter and the second would be the result of a statistical system scan conducted previous to the reductions discussed in section eight.

To continue with our operation on the original function given in Fig. 23, we see that the three combined sets above added to the immediately observed $\overset{C}{\underset{6}{5}}$ combination leaves three elements. These may be swept

together into one wholly biased term, and one combined bigate. Fig. 25 illustrates these three conditions. We now have our function expressed in six terms.



(a)

(b)

(c)

Figure 25

If we adhere to the system of expression used previously, we can write a new expression for the initial function as:

$$F = ed'c\ f'b + fba'\ e'c + fec\ d'a + fe'dcb'a + f'edc'\ ba + f'ecb'$$

Still we have not used the symmetric properties of the bigate to their fullest advantage. We know that, when employing all the input variables and one bigate, we can cover two combinations of the input variables, namely the one named and its ones complement. We have seen that we can combine the output of a bigate with a standard "and" gate, and still have two combinations reflected in the output.

Now let us reverse this configuration, and set the output of a standard gate into a bigate. See Fig. 26.

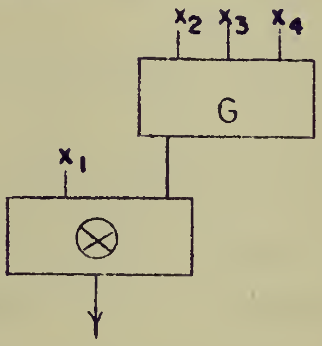


Figure 26

We see that we have a response to all four signals "up" and seven responses to x_1 down, allowing us to cover half the entire spectrum of the possible combinations of all four variables. If we held x_1 as a "control" we find that we can cover a much larger group of functions than by using the previous arrangement of hardware elements.

By arbitrary application of these concepts without mechanical means, we can operate on our input variables with the hardware configurations of Fig. 27.

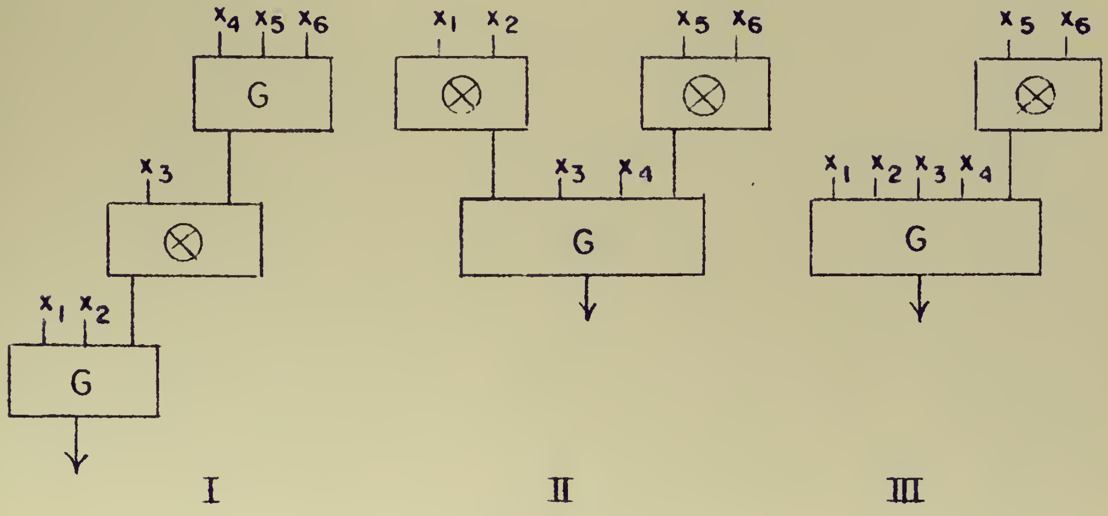


Figure 27

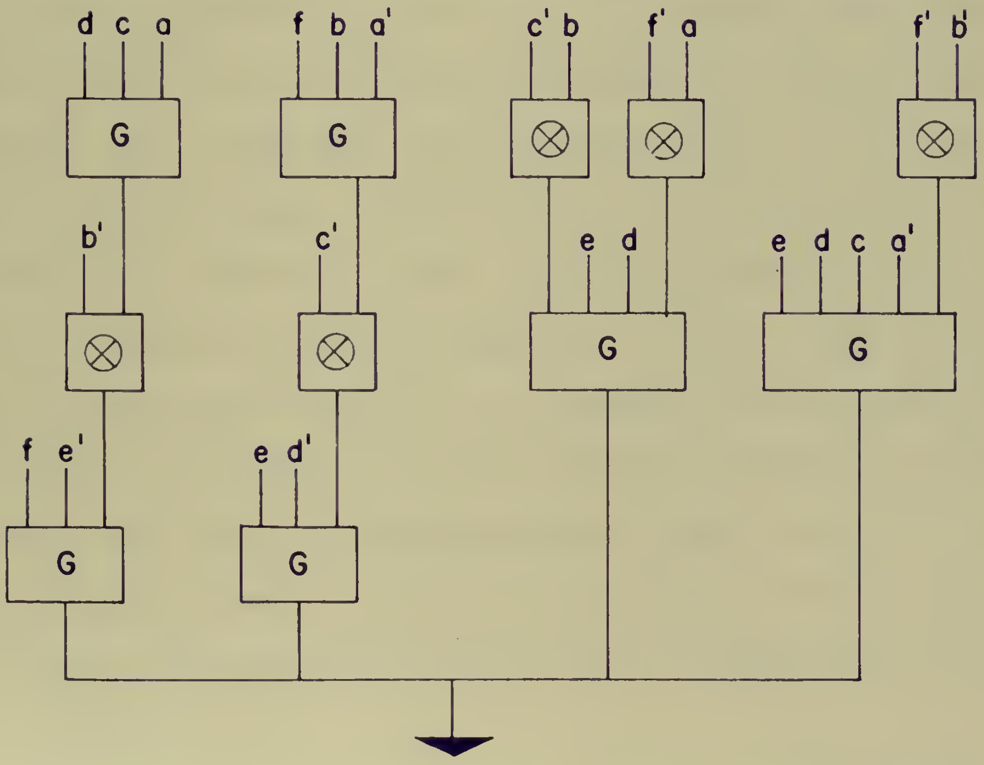
Using the roman numerals as operators and x_i as signal selection from a left to right scan, we can now write the equation for our function as:

$$F = I(fe'bdca + ed'c'fba') + II(c'bedf'a) + III(edca'f'b')$$

which allows us to use four basic terms. Since this was accomplished. without benefit of mechanical processing, the task has been somewhat simplified by the employment of only the two-input bigate, or "exclusive or" circuits. For purposes of comparison, Fig. 28(a) shows the four partitions of the function as they are ordered to make use of the configurations of Fig. 27. Fig. 28(b) is the final hardware arrangement. This is the results of applying unrefined combinatorial techniques which are based on the premise that symmetrical circuit components are available. These are still trial and error methods, but can be standardized for mechanization.

				F
0	0	0	0	0
0	0	0	0	0
0	0	0	0	0
0	0	0	0	0
0	0	0	0	0
0	F	0	0	F
0	0	1	0	1
0	0	4	8	C
3	0	0	0	3
3	0	0	0	3
3	0	0	0	3
6	0	0	0	6
0	2	0	0	2
0	D	0	0	D
0	0	2	0	2
0	0	8	2	A

(a)



$F = \mathrm{I}(fe'b'dca + ed'c'fba') + \mathrm{II}(c'bedf'a) + \mathrm{III}(edca'f'b')$

(b)

Figure 28

11. Conclusions

Although the eventual aim of researchers in the field of logical design is to produce machines capable of completely designing other machines, it is more realistic to first provide the instruments which allow man to do this job more efficiently. In the first few sections it was shown that an area of logical design that needed attention was that which deals with logical network synthesis. Numerous techniques have been devised, primarily by mathematicians, which treat certain input data and provide reduced logical networks. Unfortunately these techniques were derived using simplification criteria which are no longer wholly commensurate with the present "state of the art". Since the first Boolean reduction techniques were developed the component speeds have increased manyfold until now one megacycle germanium or silicon circuits with magnetic core and magnetic drum memories organized to operate at this speed are not uncommon. Thus the reduced networks which have been pushed by machines into two level and-or configurations do not necessarily represent an optimal arrangement. In fact, since there are so many constraints and variables to be considered, machine methods are being shunned by logical designers as inadequate in the field of reduction. Instead they rely mainly on their knowledge and ingenuities to both create and optimize their networks. Without more pronounced breakthroughs it is doubtful whether the techniques now known will be of any more than mere academic interest.

To make the problem more apparent present reduction techniques were compared and their specific inadequacies enumerated. In addition, we examined the constraints with which logical designers must deal.

A pooling of these inadequacies and constraints clearly provides the direction which research must take to produce more realistic reduction techniques. This can not be realized until we break away from the simple diode and-or criterion upon which all proponents of new methods base their logical reductions. For this reason, the investigation presented in this paper begins without defining a criterion. First of all there was no intention of producing a method which could be called a new "reduction" technique, but rather, as was stated earlier, we desired a means by which we could study the symmetric properties of response functions. In addition, it is considered premature to specify simplification criterion before we have assembled data concerning other properties and methods by which they might be combined. Finally, it is necessary to decide what configurations of real circuits should be looked on as optimum[1].

The present investigation has provided that part of the data which concerns the symmetric properties of the response function. Using a partitioning process we extracted sub-functions which had been defined as symmetries and permitted closer examination of these new properties. Because of the symmetric nature of our sub-functions the investigation was directed toward development of symmetric logical expressions. These, in turn, provided the specifications for networds using symmetric components. Since a primary objective was to provide methods which are readily mechanizable, the procedures and notation have been put in forms suitable for direct conversion to machine language. This

---

[1] Note that the term optimum is, as usual, ambiguous where undefined. The definition is purposely omitted in this case because this is as yet an unanswered question.

involved the use of hexadecimal notation which permits reasonably large functions to be handled by hand and, what is more important, greatly reduces the data processing problems of the computers and/or off-line equipments.

The techniques evolved by studying symmetrics now provide a means by which the logical designers can automatically specify network configurations beyond the restrictions of simple and-or two level circuits. Methods have been formulated which produce a three level logic using the postulated complementing device, the bigate, and in many cases the already available "exclusive or", single transistor, circuit. It might be well to mention here that this circuit, the "exclusive or", represents considerable saving of components when we consider how this operation is performed by diode and-or logic. The simplified circuit uses one transistor and several resistors, while the other method employs six diodes and a greater number of resistors. In addition, diode circuits require current boosters or amplifiers to maintain similar power levels. Thus, even though our present logical schematics using symmetric hardware may appear more complex, other factors must be considered. One of these is the fact that we cannot tell as yet what form the more general complementary bigate will take or even whether or not available components arranged to act as bigates will lead to simplified circuits. In any event we have developed a reasonable need for such components and in so doing have empirally provided specifications for hardware design to component engineers.

It is apparent that the use of symmetries alone cannot provide the answer to our simplification problems, nor can we hope to

produce simpler circuits in each case by using these techniques. We have, however, developed a tool that can, in instances, where the initial function is inherently symmetric, produce the most simple network. As an example, if we were to consider any function that was by itself what we previously referred to as sub-functions, the resulting expression in symmetric form will be one term. On the other hand, available reduction techniques would give several terms in most cases. In addition, it should be noted that the final expressions in symmetries have the characteristic expected of reduction methods in that greater symmetric cover tends to induce simpler circuits.

12. Areas recommended for future investigation and research

The results of the present investigation have suggested two basic types of further work which should be pursued to enhance our understanding of logical network synthesis. The first of these treats the broader concepts of the logical design problem from an overall systems viewpoint. As an example, we observe from reviewing the available literature that there are no clear guides which specify the constraints applicable to any particular design problem. Even after we have determined the constraints, what is their relative importance? How do we fit them together and impose them on the design? All of these questions are in most cases answered differently by individual designers. They must rely on their own knowledge and ingenuities without the advantage of formulated procedures to help them. Therefore, a study which treats with one or more of the important design constraints and evolves rules useful to logical designers is needed.

The second type problem is more restrictive in scope and tends to be more appropriate for thesis type investigations. Some problems of this type suggested by the present thesis study are described below.

A. Hardware development

Treating the symmetric properties of response functions leads us to final logical expressions containing complementary terms. To translate these expressions into hardware configurations we postulated a complementing hardware device not currently available. This device was called a "bigate" since it was capable of performing two separate complementary gating actions. There are at least two methods of approaching the problem of making these networks physically realizable. The most straight forward procedure is to produce the gating

action with a combination of conventional and-or gates, amplifiers and/or invertors as necessary. This appears to have the single advantage of providing a compact circuit useful for the very limited applications where we have a single symmetric sub-function. On the other hand, we tend to lose the inherent advantages of being able to represent portions of larger networks with a relatively simple symmetric device. Thus another approach would be to use the generalized "exclusive or" gate described in Appendix B as a stepping off point and expand its use to more than two input variables. An additional aspect of this problem should be to study means of applying biasing levels to the element itself. This would eliminate the need for the additional "and" gates.

B. Mechanization of Partitioning processes

In section six partitioning was defined as a method by which we could extract the inherent properties from the response function. Two specific methods, congruences and symmetries, have now been developed in considerable detail and in order that sufficient data can be made available for further research studies these processes should be mechanized. This can be accomplished either by directly programming a general purpose computer to carry out the rules which have been specified or by designing and constructing a special purpose data processor which can apply the definitions of the desired properties. A suggested method in the latter case would be to use combinations of read-write devices along with a drum storage. The spacing of the read heads would be determined by the order of the congruence or symmetry to be extracted. For example, if we were looking for

symmetry or congruence in the lowest order variable the heads would be separated by only one bit position. To extract higher order symmetries and congruences individual networks would be required but the same read head arrangements could be used. A data processor of this type could be sufficiently flexible to enable us to study additional properties of response functions besides congruences and symmetries.

C. Define and examine other properties of the response function

Another form of symmetry is suggested by the neat package of "exclusive or" gates which can be used to represent the sum digit of a binary adder. The adder must form a logical sum of three digits and is best shown by the following table:

Input digits			Sum digit	
c	b	a	S	
0	0	0	0	
0	0	1	1	6
0	1	0	1	
0	1	1	0	
1	0	0	1	
1	0	1	0	9
1	1	0	0	
1	1	1	1	

Table 2

The signal S may be expressed mathematically by the following logical equation:

$$S = (a \oplus b) \oplus c$$

This equation indicates that the sum signal S may be formed by combining

the signals "a" and "b" in accordance with the "exclusive or" operation and then combining this output in a second "exclusive or" gate with "c". This is shown below.

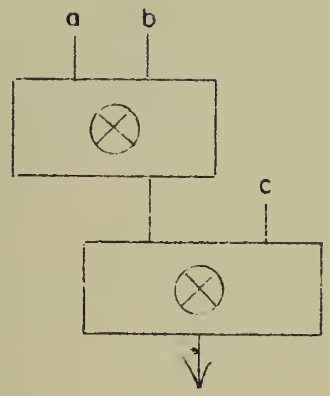


Figure 1-

In table one we observe that the signal S can be represented hexadecimally as $\underset{9}{6}$. This is a property which we could extract by complementing the bottom four bits of the function and then folding upward to recover the normal "c" symmetry.

D. Sequential operations.

Still another property can be introduced if we remove the parallel operating restrictions and spread the problem in time. This might best be studied by defining a time varying control function which could be aligned alongside the normal response function. The frequency which with this control function changed would have to be in excess of the natural bit rate to permit a sequence of operations on the input variables during each bit time. As the control function changed it would designate the allowable input configurations; all

others will be redundant. This amounts to time sharing circuit elements since the gates would be active for the particular configuration only during a specified segment of a bit time. Processing the data in a controlled sequential manner such as this is analogous to the use of control functions in DDA Operations. In both cases the control function is used to establish the time pattern of operations.

BIBLIOGRAPHY

1. D. D. McCracken, Digital Computer Programming, John Wiley & Sons, Inc., 1957.

2. G. Boole, An Investigation of the Laws of Thought, Cambridge, England, 1854, Dover Publications, New York, 1954.

3. C. E. Shannon, Symbolic Analysis of Relay and Switching Circuits, A.I.E.E. Transactions, 57, pp. 713-723, 1938.

4. R. H. Urbano & R. K. Mueller, A Topological Method for the Determination of the Mineral Forms of Boolean Functions, IRE Transactions on Electronic Computers, EC-5, September 3, 1956.

5. W. V. Quine, The Problem of Simplifying Truth-Functions, American Mathematical Monthly, 59, pp. 521-531, 1952.

6. W. V. Quine, A Way to Simplify Truth Functions, American Mathematical Monthly, 62, pp. 627-631, 1955.

7. E. J. McCluskey, Jr. Minimization of Boolean Functions, Bell System, Technical Journal, 35, pp. 1417, 1956.

8. E. W. Veitch, A Chart Method for Simplifying Truth Functions, Proc. Association for Computing Machinery Conference, pp. 127-133, May 2-3, 1952.

9. R. J. Nelson, Simplest Normal Truth Functions, J. Symbolic Logic, 20, number 2, pp. 105-108, 1955.

10. R. J. Nelson, Weak Simplest Normal Truth Functions, J. Symbolic Logic, 20, Number 3, pp. 232-234, 1955.

11. Staff Harvard Computation Laboratory, Synthesis of Electronic Computing and Control Circuits, Harvard University Press, Combridge, 1951.

12. Shreeram Abhyankar, Minimal "Sum of Products of Sums" Expressions of Boolean Functions, IRE Transactions on Electronic Computers, EC - 7, 4, December, 1958.

13. Shreeram Abhyankar, Absolute Minimal Expressions of Boolean Functions, IRE Transactions on Electronic Computers, EC - 8, 1 March, 1959.

14. T. C. Bartee, The Automatic Design of Logical Networks, Lincoln Laboratory Technical Report, No. 191, December 3, 1958.

15. Montogomery Phister, Jr., Logical Design of Digital Computers, John Wiley & Sons, Inc., 1958.

16. M. Karnaugh, The Map Method of Synthesis of Combinational Logic Circuits, Communications and Electronics, pp. 593-599, November, 1953.

17. R. K. Richards, Arithmetical Operations in Digital Computers, D. Van Nostrand Co., 1955.

18. J. R. Logan, Logic Reduction as a Partitioning Process, Litton Industries, Canoga Park, California, Unpublished.

19. J. R. Logan, A Computer Method for Network Design, Litton Industries, Beverly Hills, California, Unpublished.

20. F. Horner, Some Mathematical Aspects of Switching, American Mathematical Monthly, 62, pp. 75-90, February 1955.

APPENDIX A

POSTULATES AND IDENTITIES OF BOOLEAN ALGEBRA

A. 1 The postulates of boolean algebra [20]

(1a) $a + b = b + a$ Commutative laws

(1b) $ab = ba$

(2a) $a + (b + c) = (a + b) + c$ Associative laws

(2b) $a(bc) = (ab)c$

(3a) $a(b + c) = ab + ac$ Distribution laws

(3b) $a + bc = (a + b)(a + c)$

(4a) $a + a = a$ The Idempotent laws

(4b) $a \cdot a = a$

(5a) $0 + a = a$ The laws of operation with zero

(5b) $0 \cdot a = 0$

(6a) $1 + a = 1$ The laws of operation with one

(6b) $1 \cdot a = a$

(7a) $a + a' = 1$ The laws of complementarity

(7b) $a \cdot a' = 0$

(8a) $(ab)' = a' + b'$ The dualization laws

(8b) $(a + b)' = a'b'$

(9) $(a')' = a$ The law of involution

(10) $a \oplus b = b \oplus a$

(11) $a \oplus a = 0$

(12) $a \oplus 0 = a$

(13) $a \oplus a' = a' \oplus a = 1$

(14a) $a \oplus 1 = a'$

(14b) $1 \ominus a = a$

A. 2 Useful boolean identities

(a) $a + ab = a$

(b) $a + a'b = a + b$

(c) $a(a' + ab) = ab$

(d) $a(a + b) = a$

(e) $a(a' + b) = ab$

(f) $(a + b)(a' + c)(b + c) = (a + b)(a' + c)$

(g) $ac + a'b + bc = ac + a'b$

(h) $(a + b)(a' + c) = ac + a'b$

(i) $a \oplus b = a'b + ab'$

(j) $a_1 \oplus a_2 \oplus \cdots \oplus a_n = \sum$ all products with an odd number of true terms

(k) $a \oplus b \oplus c = a/b/c$

APPENDIX B

TRANSISTORIZED "EXCLUSIVE OR" GATE

Several years ago the "exclusive or" transistor circuits of Fig. 1(a) and 1(b) were developed, patented and placed in use at Litton Industries, Beverly Hills, California.

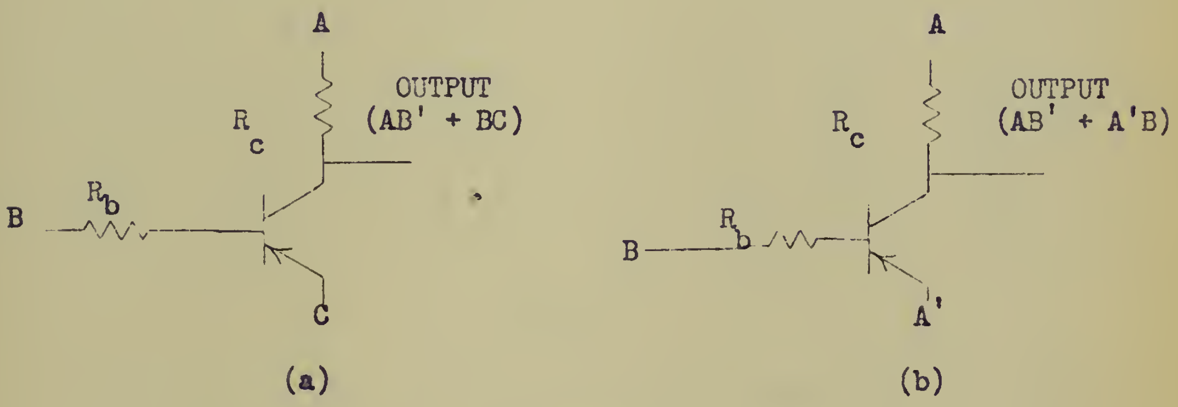


Figure 1

The circuit of Fig. 1(a) handles three unrelated signals and is called the standard "exclusive or". Fig. 1(b) uses the false state of "A" as an input vice "C" and is considered a special case of Fig. 1(a). These circuits can be cascaded directly to form a chain of "exclusive or" circuits. However, when they feed into diode gates current boosters should be used to act as buffers between the "exclusive or" and diode logic. This is necessary since the output from either of the two circuits shown is partially degraded in voltage and power from the levels of the inputs.

The manner in which the output signal is produced and the precise nature of the relationships existing between the levels of the output signal and the levels of input signals can be clarified by considering the operation of the basic transistor circuit. This must be

done for each possible combination of levels of the bilevel input signals A, B and C of Fig. 1 (a). These combinations are all given in Table 1 along with the level of the output produced in each case.

Combination	Level of input signals			Level of output signal assuming the transistor is	
	C	B	A	PNP	NPN
1	0	0	0	0	0
2	0	0	1	0	1
3	0	1	0	0	0
4	0	1	1	1	0
5	1	0	0	1	0
6	1	0	1	1	1
7	1	1	0	0	1
8	1	1	1	1	1

Table 2

The following description is for the PNP transistor. A similar treatment considering the transistor as an NPN type produces an output given by the extreme right column of Table 1.

Combination 1. Since there is no source of a higher voltage level, the output signal must also be at a low level.

Combination 2. A high, B and C low. Since signal A is high and signal B is low, the P-N collector-base junction will be forward biased, causing it to be highly conductive. Consequently the collector and base will be at nearly the same voltage level. A current I_c flows across the junction from the collector into the base and a nearly equal current flows from base to emitter. The resultant base current $I_b = I_c - I_e$ is nearly zero, hence no appreciable voltage drop can occur across the base resistor, R_b. Thus the base and collector are both at the low level of B.

Combination 3. A and C low, B high. In this case both junctions are back biased so that they are non-conducting. Hence the output signal assumes the level of the signal C which is low.

Combination 4. A and B high, C low. The emitter-base junction is back-biased, non conducting and there is no effective bias across the collector-base junction so that it too is substantially non-conductive. Since no current is drawn through the collector the output signal at the collector corresponds to A and is high.

Combination 5. A and B low, C high. The emitter-base junction is forward-biased conductive so that the base assumes the high level of the emitter, C. Because of inherent transistor action current flows across the back-biased collector-base junction and the collector assumes the high voltage level of the base.

Combination 6. A and C high, B low. Both junctions will be forward biased, thus the voltage level of the base corresponds to the high level of the emitter. This high level is carried over to the collector by the highly conductive emitter-base junction.

Combination 7. B and C high, A low. The collector-base junction is back-biased non-conducting while the emitter-base junction is essentially unbiased. Therefore, the collector or output assumes the low level of A.

Combination 8. A, B, and C high. Since there is no source of low voltage, the output is high.

Fig. 1(b) depicts the same circuit just described but with A' replacing C as the signal being applied at the emitter. If we make this substitution into Table 1 it is apparent that the output is the

"exclusive or" function.

An important feature of this transistor gate is that we realize a considerable hardware reduction whenever the output function of Fig. 1(a), $AB' + BC$, or other adaptations of this function are mechanized. Using the triple input transistor circuit enables us to product the output involving the false state of B without providing this level of B among the inputs. If we mechanize to obtain the output with and-or logic alone it is necessary to provide all signals which occur as part of the output expression. In those cases where the complement is not readily available it necessitates adding an inverting amplifier to form the signal B'. To complete the mechanization we must use two "and" gates and one "or" gate each comprising two diodes and one resistor. Thus we need one inverting amplifier, six diodes and three resistors by prior art techniques but only a single transistor and a few resistors by the triple input transistor gate.

Note also that it is a simple matter to obtain the "equal or" function mentioned in section six merely by making the following substitutions in Fig. 1(a). Let $A = A'$, $B = B$ and $C = A$. The output becomes $A'B' + AB$.

APPENDIX C

THREE BY THREE BIT MULTIPLIER

The response functions of the following three-by three bit multiplier are used as arbitrary functions to show further examples of symmetric partitioning. The multiplier is "cba" and the multiplicand is "fed" while F, E, D, C, B and A special y hexadecimally the six 64 bit response functions. The "D" function

$$\begin{array}{cccccc} & & & f & e & d \\ & & & c & b & a \\ \cline{4-6} & & & fa & ea & da \\ & & fb & eb & db & \\ & fc & ec & dc & & \\ \cline{2-6} \end{array}$$

F	E	D	C	B	A
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	3	5
0	0	0	F	3	5
0	0	0	3	5	0
0	0	F	3	5	0
0	0	1	2	6	5
0	3	C	D	6	5
0	0	3	5	0	0
0	F	3	5	0	0
0	0	3	5	3	5
1	E	6	A	3	5
0	1	2	6	5	0
3	C	D	6	5	0
0	1	2	7	6	5
7	9	A	8	6	5

should be recognized as the function treated in sections six through nine.

We can now partition the remaining functions into symmetric functions, and, where these do not provide complete cover, a characteristic noise function, δ . The only truly symmetric

function as defined in section six is the "E" response. In this case there is no noise function. To permit easy reference the following tables include both the symmetric sub functions and congruences. The symmetries are tabulated down the left half of the page while the congruences are to the right. Note that the "A" function is completely asymmetric and consequently must be represented as all noise in the symmetric system. Since each column and the individual tables specify the degree to which a function is either congruent or symmetric in that input variable, the absence of data in a column means the function is non-congruence or asymmetric in that variable.

A Function

Initial Function	Symmetries							Congruences						
	f	e	d	c	b	a	S	f	e	d	c	b	a	S
0							0	0	0		0	0		
0							0	0	0		0	0		
5							5	5	5		5	5		
5							5	5	5		5	5		
0							0	0	0		0	0		
0							0	0	0		0	0		
5							5	5	5		5	5		
5							5	5	5		5	5		
0							0	0	0		0	0		
0							0	0	0		0	0		
5							5	5	5		5	5		
5							5	5	5		5	5		
0							0	0	0		0	0		
0							0	0	0		0	0		
5							5	5	5		5	5		
5							5	5	5		5	5		
16							16	16	16		16	16		bit count

B Function

Initial function	Symmetries							Congruencies							
	f	e	d	c	b	a	δ	f	e	d	c	b	a	δ	
0	0	0	0	0	0	0	0	0	0	0	0	0	0		
0	0	0	0	0	0	0	0	0	0	0	0	0	0		
3	2	2	0	0	0	3	0	3	2	0	3	0	3		
3	2	2	0	0	0	3	0	3	2	0	3	0	3		
5	4	4	4	0	0	0	1	5	0	4	5	5	0		
5	4	4	4	0	0	0	1	5	0	4	5	5	0		
6	0	0	2	6	6	0	0	6	2	4	6	0	0		
6	0	0	2	6	6	0	0	6	2	4	6	0	0		
0	0	0	0	0	0	0	0	0	0	0	0	0	0		
0	0	0	0	0	0	0	0	0	0	0	0	0	0		
3	2	2	0	0	0	3	0	3	2	0	3	0	3		
3	2	2	0	0	0	3	0	3	2	0	3	0	3		
5	4	4	4	0	0	0	1	5	0	4	5	5	0		
5	4	4	4	0	0	0	1	5	0	4	5	5	0		
6	0	0	2	6	6	0	0	6	2	4	6	0	0		
6	0	0	2	6	6	0	0	6	2	4	6	0	0		
24	8	8	8	8	8	8	4	24	8	8	24	8	8	0	bit count

C Function

Initial function	Symmetries							Congruencies							
	f	e	d	c	b	a	δ	f	e	d	c	b	a	δ	
0	0	0	0	0	0	0	0	0	0	0	0	0	0		
0	0	0	0	0	0	0	0	0	0	0	0	0	0		
0	0	0	0	0	0	0	0	0	0	0	0	0	0		
F	6	C	0	0	F	F	0	A	D	0	0	F	F		
3	1	3	3	0	0	3	0	2	0	2	3	0	3		
3	2	0	0	0	0	3	0	2	0	1	3	0	3		
2	2	0	0	0	0	0	0	2	0	2	0	0	0		
D	8	0	C	0	9	C	0	8	D	1	0	5	C		
5	1	1	5	0	0	0	0	0	4	5	5	5	0		
5	4	4	0	0	0	0	1	0	4	0	5	5	0		
5	4	4	0	5	0	0	0	0	5	5	0	5	0		
A	8	2	A	A	0	0	0	A	8	0	0	A	0		
6	6	4	0	6	6	0	0	2	4	6	6	0	0		
6	0	2	6	6	6	0	0	2	4	0	6	0	0		
7	0	2	6	1	6	3	0	2	5	6	0	5	3		
8	0	8	0	8	0	0	0	8	8	0	0	0	0		
28	12	12	12	10	12	12	1	12	16	12	13	16	12		bit count

D Function

Initial function	Symmetries							Congruencies							
	f	e	d	c	b	a	δ	f	e	d	c	b	a	δ	
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
F	C	0	8	0	F	F	0	D	0	C	0	F	F	0	
1	0	0	1	1	0	0	0	0	0	0	0	0	0	1	
C	C	0	0	8	0	C	0	8	0	C	0	0	0	0	
3	3	1	2	0	0	3	0	0	2	3	3	0	0	0	
3	0	0	0	0	0	3	0	0	1	2	3	0	0	0	
3	3	3	0	2	0	3	0	0	2	3	2	0	0	0	
6	0	4	4	4	6	0	0	0	2	2	2	0	0	4	
2	0	2	0	2	0	0	0	0	2	2	0	0	0	0	
D	0	C	4	4	9	C	0	D	1	8	0	5	C	0	
2	0	0	2	0	0	0	0	0	2	2	2	0	0	0	
A	0	8	0	0	0	0	2	8	2	8	2	A	0	0	
22	8	8	6	6	8	14	1	8	8	14	8	8	6	2	bit count

E Function

Initial function	Symmetries							Congruencies							
	f	e	d	c	b	a	δ	f	e	d	c	b	a	δ	
0		0	0	0	0	0		0	0	0	0	0	0		
0		0	0	0	0	0		0	0	0	0	0	0		
0		0	0	0	0	0		0	0	0	0	0	0		
0		0	0	0	0	0		0	0	0	0	0	0		
0		0	0	0	0	0		0	0	0	0	0	0		
0		0	0	0	0	0		0	0	0	0	0	0		
0		0	0	0	0	0		0	0	0	0	0	0		
3		0	0	0	0	3		1	0	0	0	0	3		
0		0	0	0	0	0		0	0	0	0	0	0		
F		8	0	0	F	F		0	C	E	0	F	F		
0		0	0	0	0	0		0	0	0	0	0	0		
E		8	0	0	6	C		0	8	E	0	A	C		
1		1	1	1	0	0		0	0	1	0	0	0		
C		0	8	8	0	C		0	C	8	0	0	C		
1		1	1	1	0	0		0	0	1	1	0	0		
9		0	8	8	9	0		1	8	8	1	0	0		
15		4	4	4	8	10		2	6	10	2	6	10		bit count

F Function

Initial Function	Symmetries							Congruencies						
	f	e	d	c	b	a		f	e	d	c	b	a	
0					0	0	0		0	0		0	0	
0					0	0	0		0	0		0	0	
0					0	0	0		0	0		0	0	
0					0	0	0		0	0		0	0	
0					0	0	0		0	0		0	0	
0					0	0	0		0	0		0	0	
0					0	0	0		0	0		0	0	
0					0	0	0		0	0		0	0	
0					0	0	0		0	0		0	0	
0					0	0	0		0	0		0	0	
0					0	0	0		0	0		0	0	
1					0	0	1		1	0		0	0	
0					0	0	0		0	0		0	0	
3					0	3	0		0	3		0	3	
0					0	0	0		0	0		0	0	
7					6	3	0		1	3		5	3	
6					2	4	1							bit count